更炫酷的特效

3D游戏精彩范例



Android **3D** 全新登场！

书中着色器完全兼容

iPhone

Windows Phone

MeeGO PSP

PlayStation

Android 3D 游戏开发技术宝典

——OpenGL ES 2.0

吴亚峰 著　百纳科技 审校

视频与源程序

第一本深入探讨 OpenGL ES 2.0 3D 开发的书

- 更强大的 OpenGL ES 2.0 3D 应用，更真实、更炫酷的特效开发，更多不容错过的 3D 游戏精彩范例
- 基于 Android 平台开发，其他主流的移动平台（如 iPhone、Windows Phone、MeeGO、PSP、PlayStation 等）下 3D 应用的高级开发都可以借鉴学习
- 3 个大型的获奖 3D 游戏案例，总代码量接近 6 万行
- 5 个多小时的案例视频及所有案例源程序，帮助读者尽快进入实战角色

人民邮电出版社
POSTS & TELECOM PRESS

VISIT…

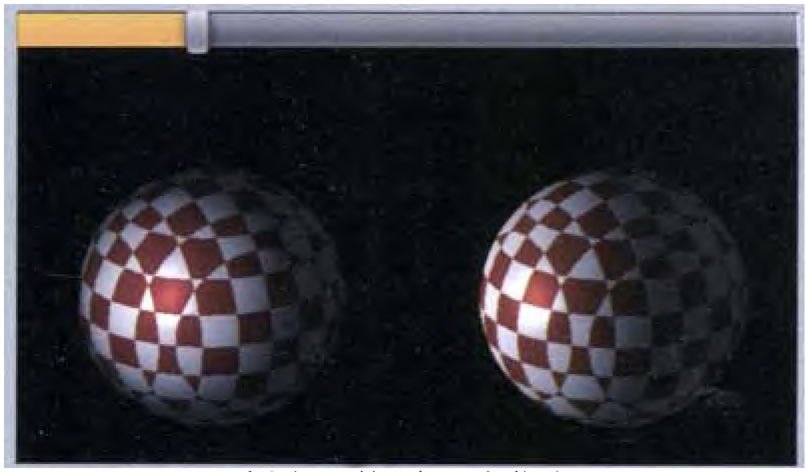
LANZAROTE Caliente.com



吴亚峰

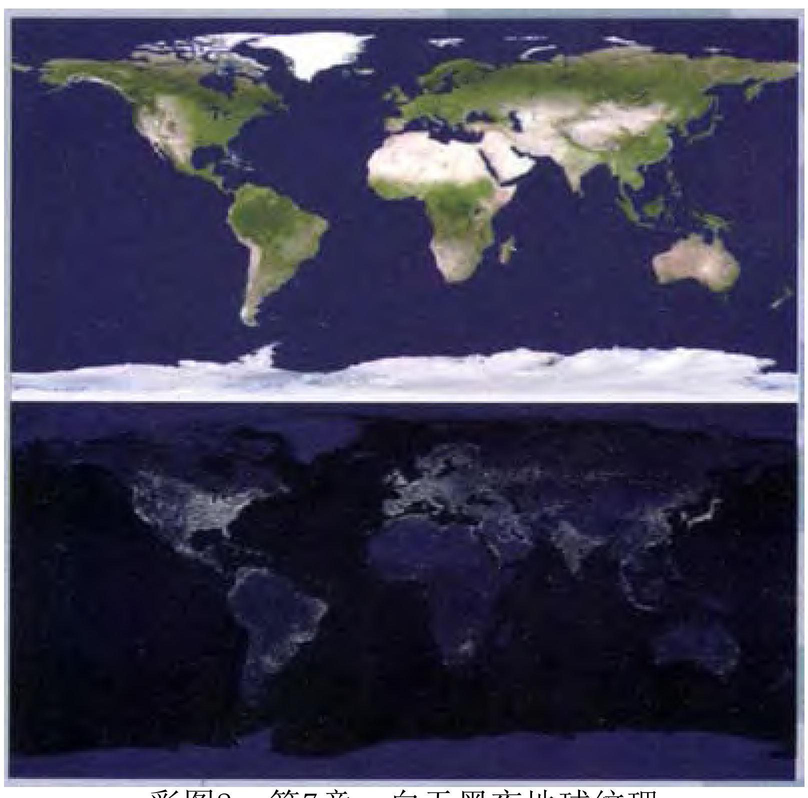
毕业于北京邮电大学，后留学澳大利亚卧龙岗大学取得硕士学位。1998年开始从事Java应用的开发，有十多年的Java开发与培训经验。主要研究方向为OpenGL ES、手机游戏、Java EE以及搜索引擎。同时为手机游戏、Java EE独立软件开发工程师，并兼任百纳科技Java培训中心首席培训师。

近十年来为数十家著名企业培养了上千名高级软件开发人员，曾编写过《Java SE 6.0编程指南》、《菜鸟成长之路——Java程序员职场全攻略》、《Android2.0游戏开发实战宝典》、《Android核心技术与实例详解》、《Android应用案例开发大全》、《Android游戏开发大全》、《Android应用开发完全自学手册——核心技术、传感器、2D/3D、多媒体与典型案例》等多本畅销技术书籍。2008年初开始关注Android平台下的应用开发，并开发出一系列优秀的Android应用程序与游戏。



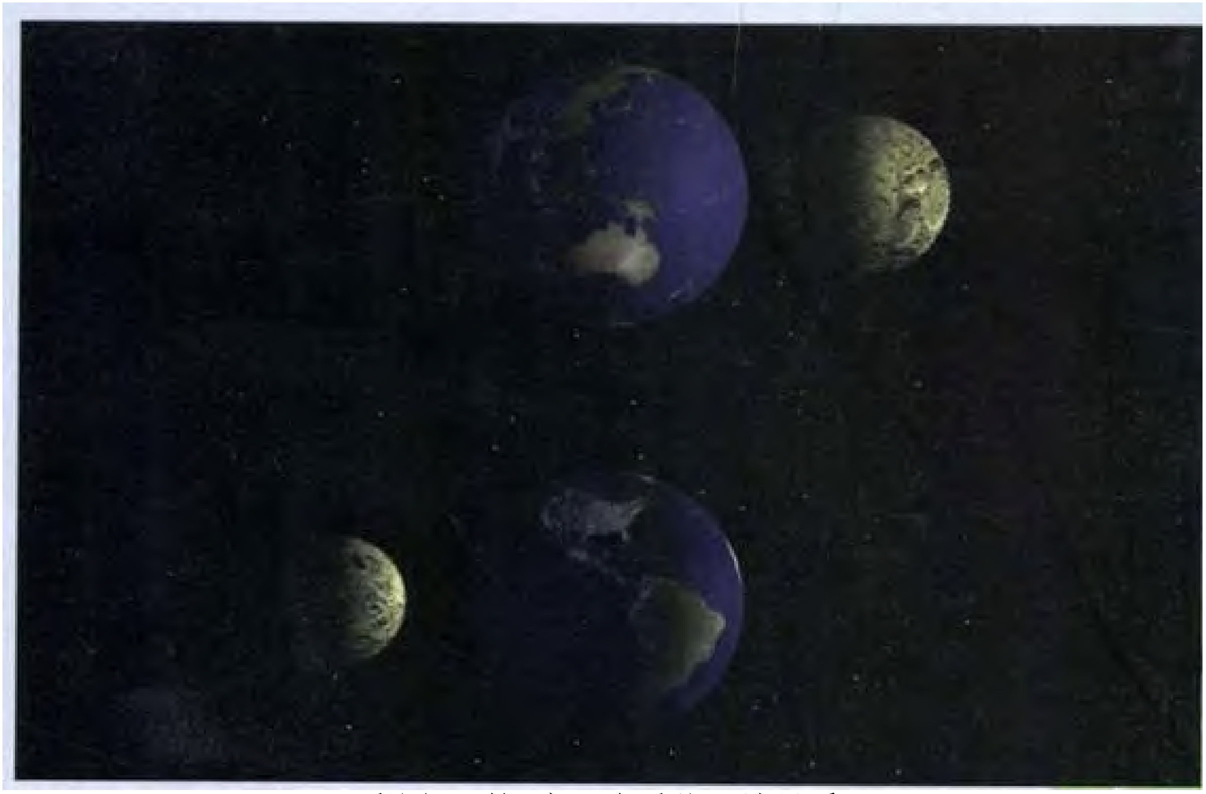
彩图1　第6章　定位光

关键技术：　用着色器实现光照计算模型包括：环境光、散射光、镜面光三个通道



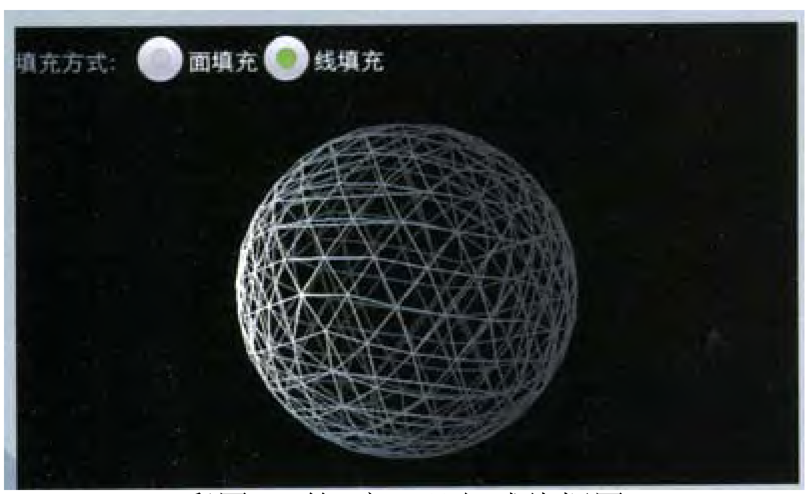
彩图2　第7章　白天黑夜地球纹理

关键技术：　多重纹理技术



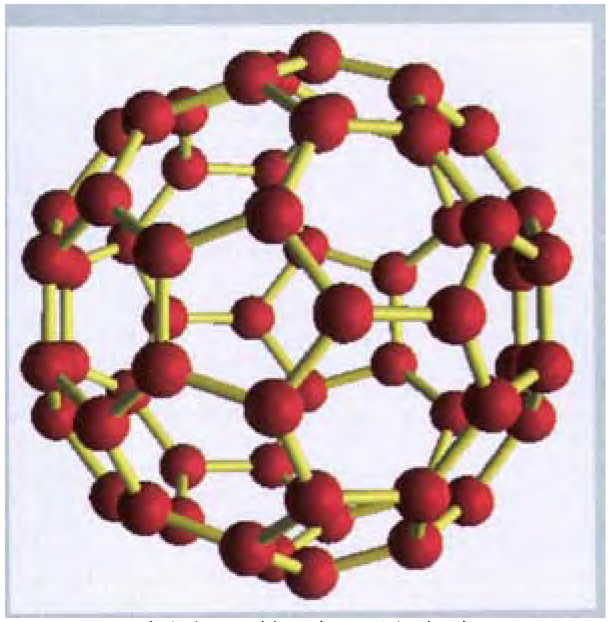
彩图3　第7章　多重纹理地月系

关键技术：　光照模型的使用、多重纹理的使用



彩图4　第8章　几何球线框图

关键技术：　几何球顶点坐标的计算、线框绘制的使用、光照的使用



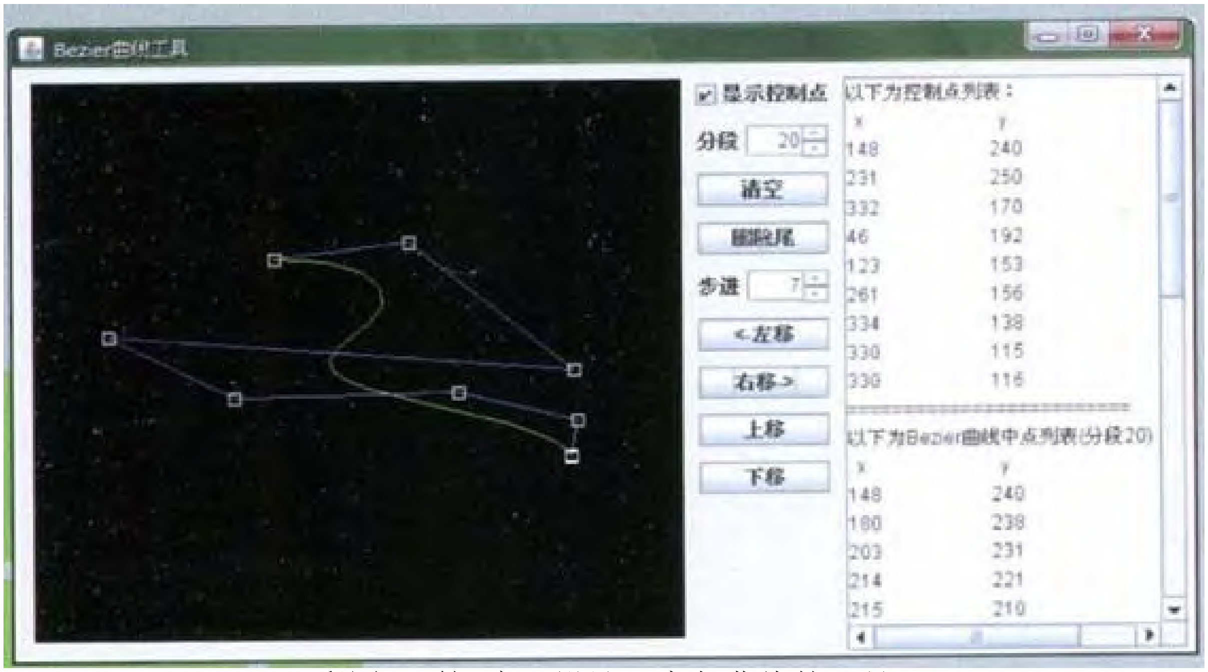
彩图5　第8章　足球碳

关键技术：　足球碳顶点的生成算法、平移变换旋转变换的组合使用



彩图6　第9章　带纹理的茶壶

关键技术：　从3DMax设计的模型中加载顶点与纹理信息



彩图7　第8章　设计贝塞尔曲线的工具

关键技术： 贝塞尔曲线生成算法的计算机编程实现



彩图8　第10章　带云层的地月系

关键技术： 光照模型的使用、多重纹理的使用、混合的使用



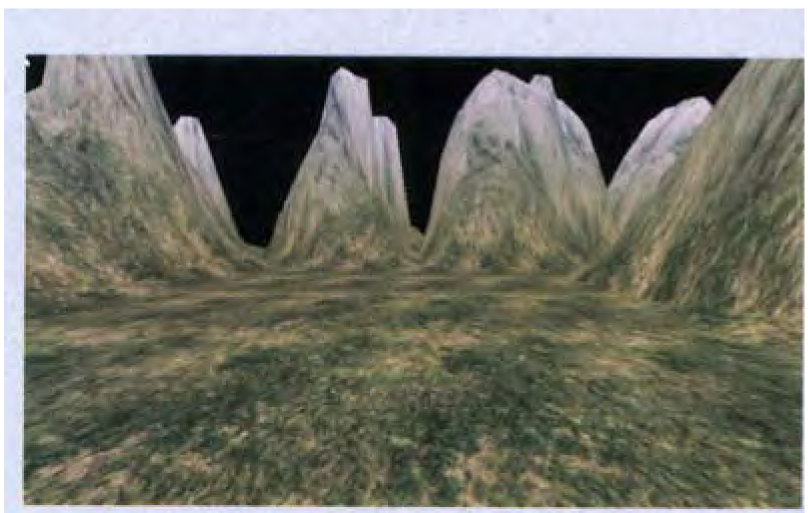
彩图9　第10章　两种雾效果的对比

关键技术： 基于着色器实现线性与非线性雾模型



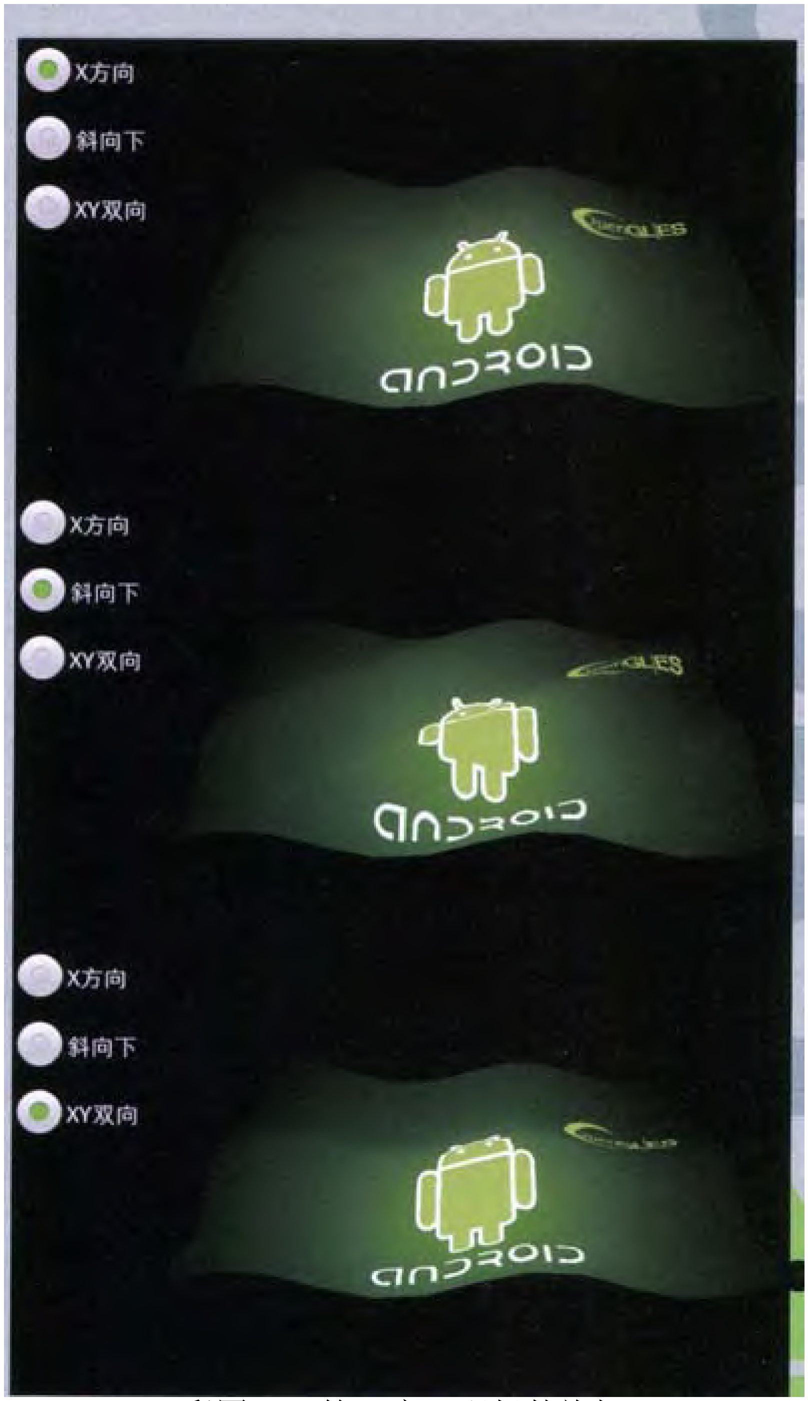
彩图10　第11章　程序纹理地形

关键技术：　灰度图地形、程序纹理与多重纹理



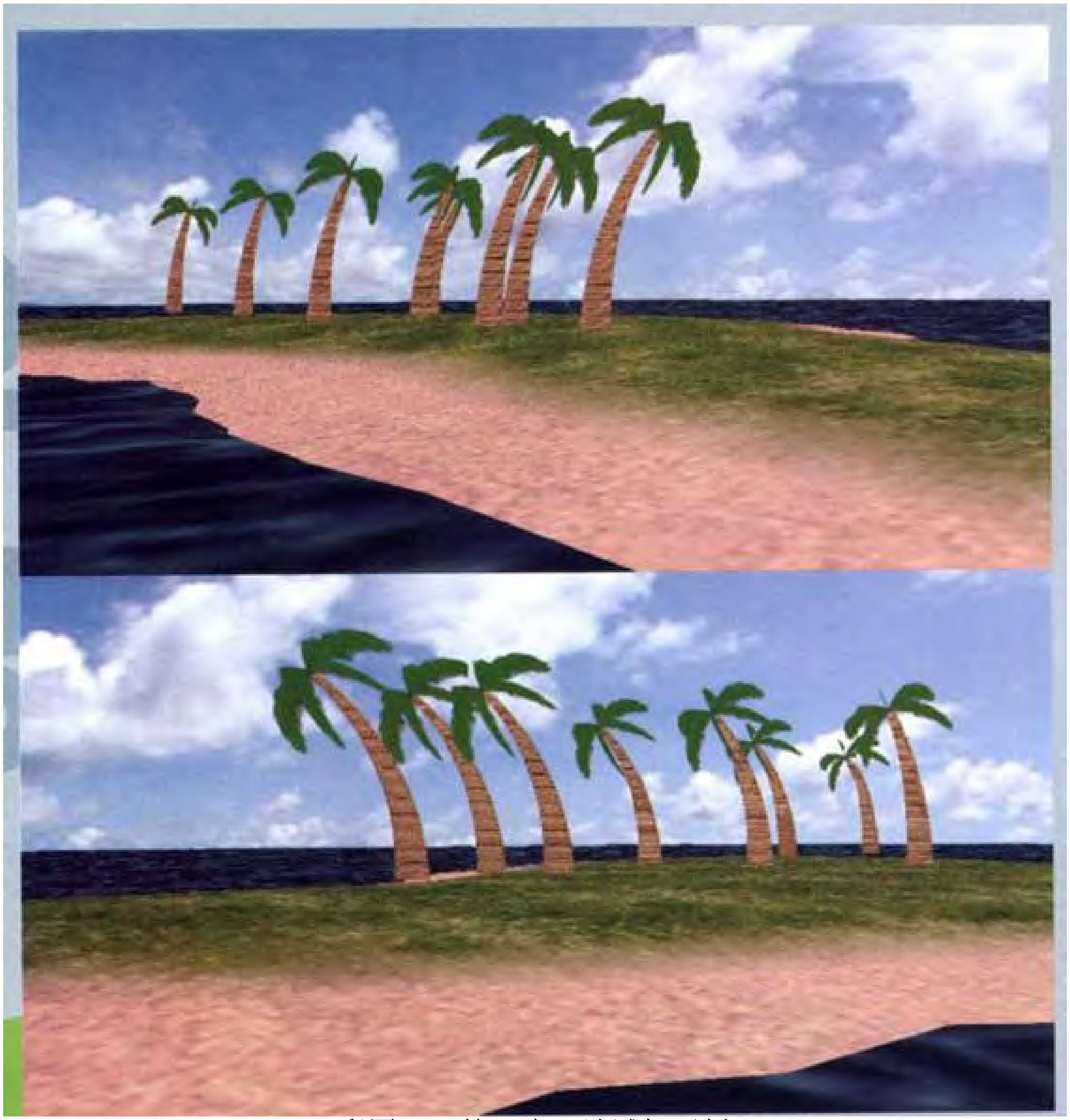
彩图11　第11章　MipMap纹理地形

关键技术：　灰度图地形、MipMap纹理、程序纹理与多重纹理



彩图12　第13章　飘扬的旗帜

关键技术： 用顶点着色器实现软体



彩图13　第13章　沙滩椰子树

关键技术：　用顶点着色器实现可编程的交互式软体



彩图14　第13章　展翅飞翔的雄鹰

关键技术： 用顶点着色器实现关键帧动画



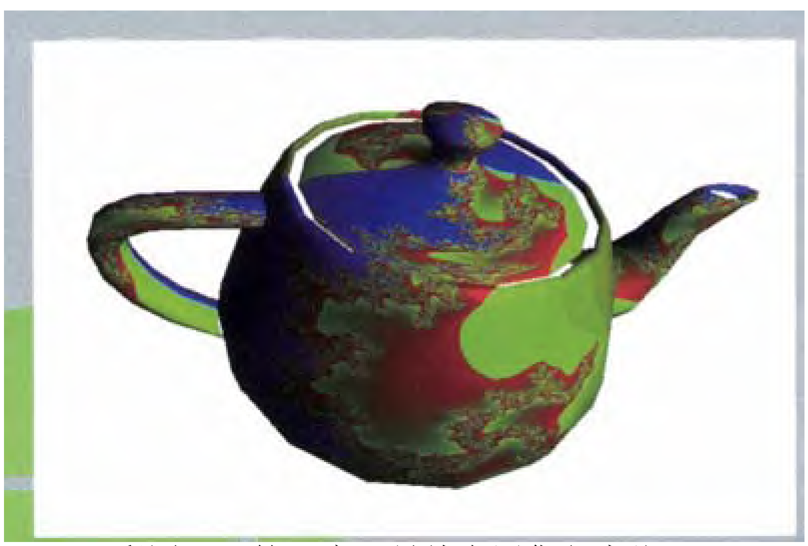
彩图15　第13章　扭动的三角形

关键技术：　用顶点着色器实现二维变形



彩图16　第14章　用着色器进行数字图像处理

关键技术：　用片元着色器基于GPU实现数字图像处理滤镜



彩图17　第14章　曼德布罗集程序纹理

关键技术：　用片元着色器实现基于曼德布罗集分形的程序纹理



彩图18　第14章　茱莉亚集程序纹理

关键技术：　用片元着色器实现基于茱莉亚集分形的程序纹理



彩图19　第15章　投影贴图

关键技术：　投影光学效果的模拟


彩图20　第15章　反射环境模拟

关键技术：　反射光学效果的模拟、立方图纹理的使用


彩图21　第15章　色散效果模拟

关键技术：　折射效果与色散效果的模拟



彩图22　第15章　菲涅尔效果模拟

关键技术：　折射与反射综合的菲涅尔效果的实现



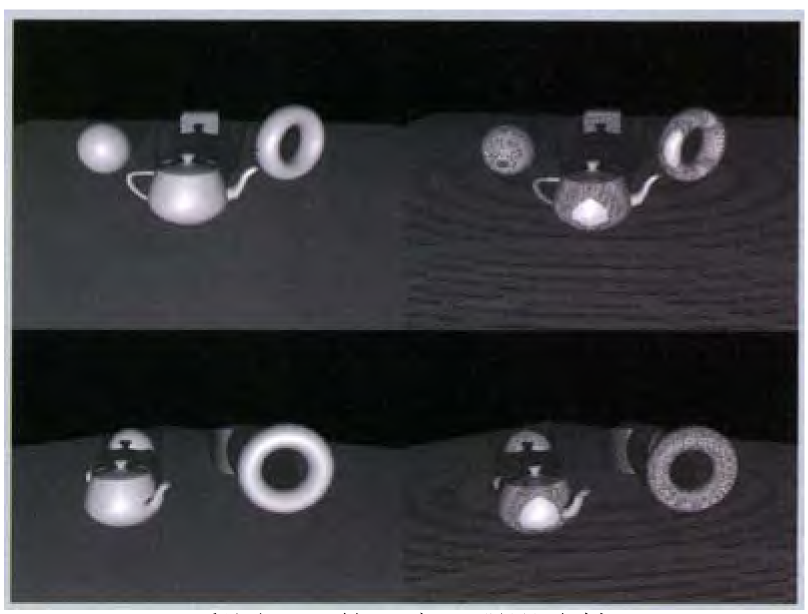
彩图23　第15章　凹凸映射

关键技术：　用片元着色器与法向量纹理图结合实现凹凸映射



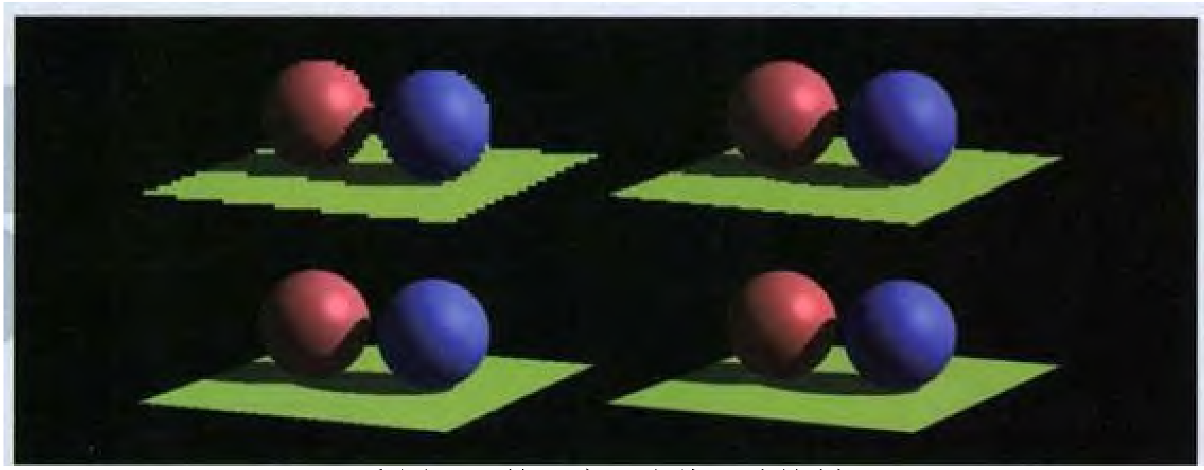
彩图24　第15章　距离纹理

关键技术：　服务于阴影映射技术的距离纹理生成算法的使用



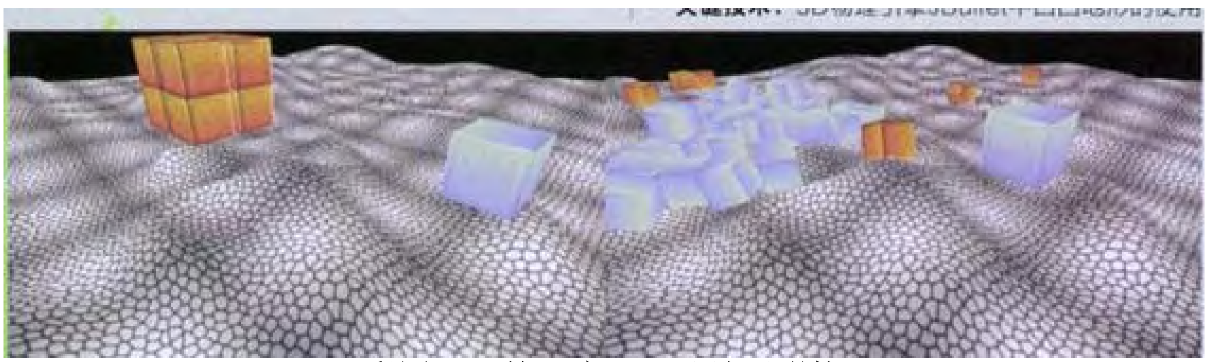
彩图25　第15章　阴影映射

关键技术：　基于阴影映射技术的立体阴影实现



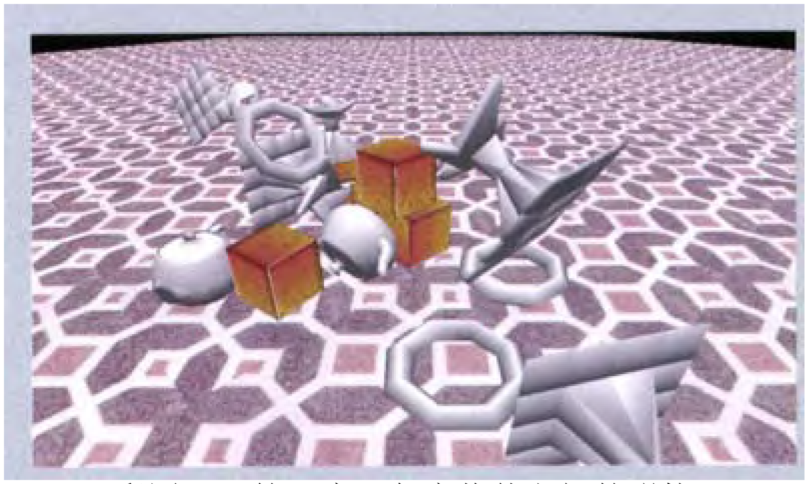
彩图26　第15章　光线跟踪绘制

关键技术：　光线跟踪算法的实现



彩图27　第17章　凹凸地形碰撞

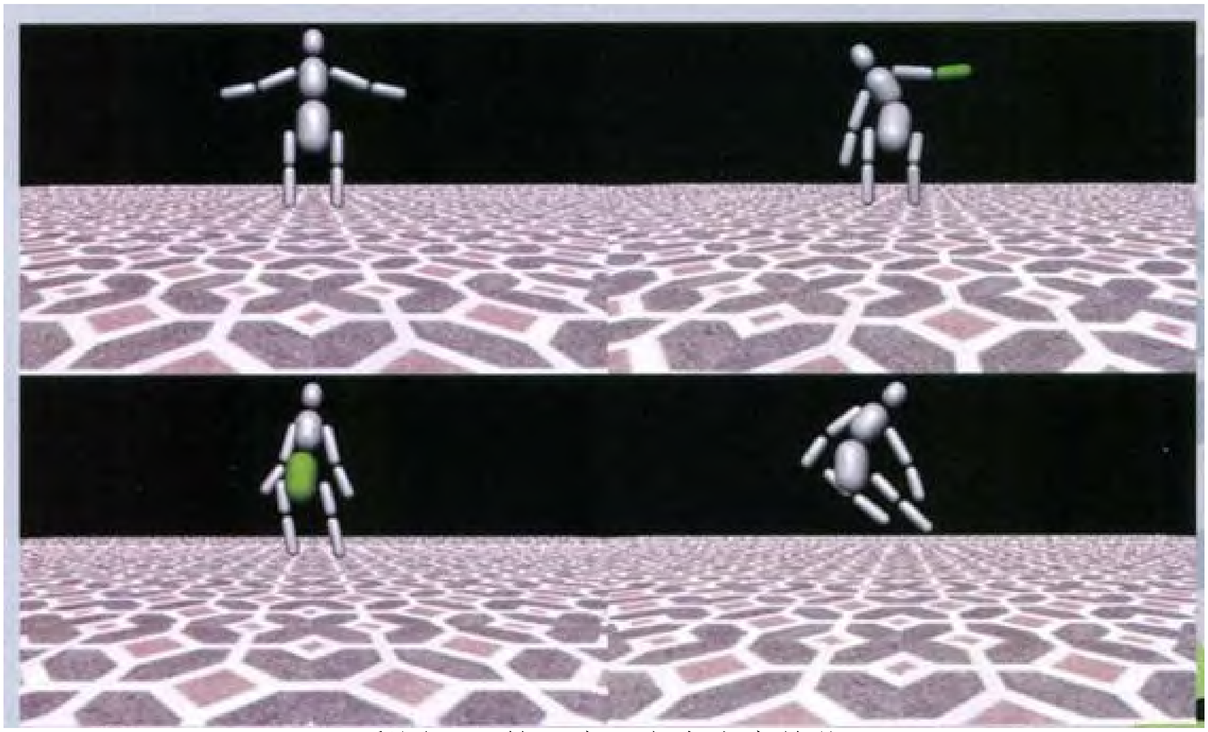
关键技术：　3D物理引擎JBullet中凹凸地形的使用


彩图28　第17章　任意物体之间的碰撞

关键技术：　3D物理引擎JBullet中任意形状物体技术的使用


彩图29　第18章　重力小球

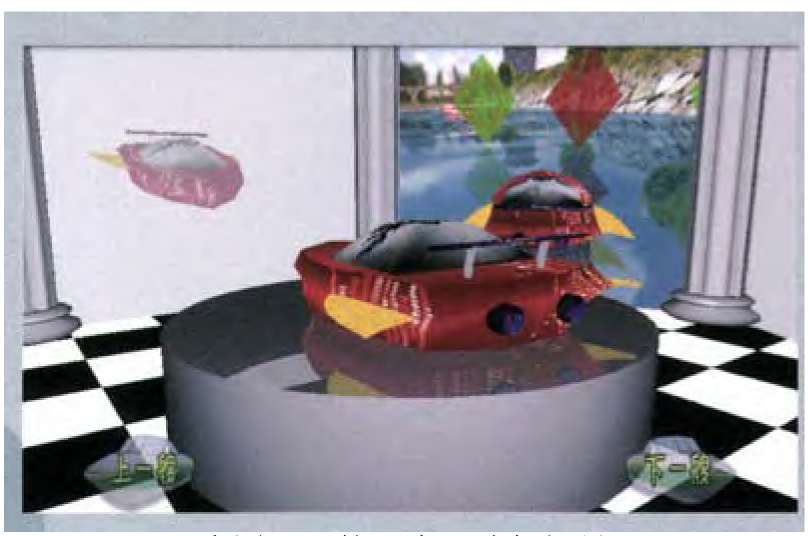
关键技术：　用姿态传感器进行体感操控算法的使用



彩图30　第17章　六自由度关节

关键技术：　3D物理引擎JBullet的使用、3D拾取技术的使用

BN赛艇案例开发亮点：　大型凹凸地形的生成、物体与任意地形的碰撞检测、倒影及后视镜特效的开发。



彩图31　第20章　选艇场景



彩图32　第20章　计时模式游戏场景



彩图33　第20章　竞速模式游戏场景

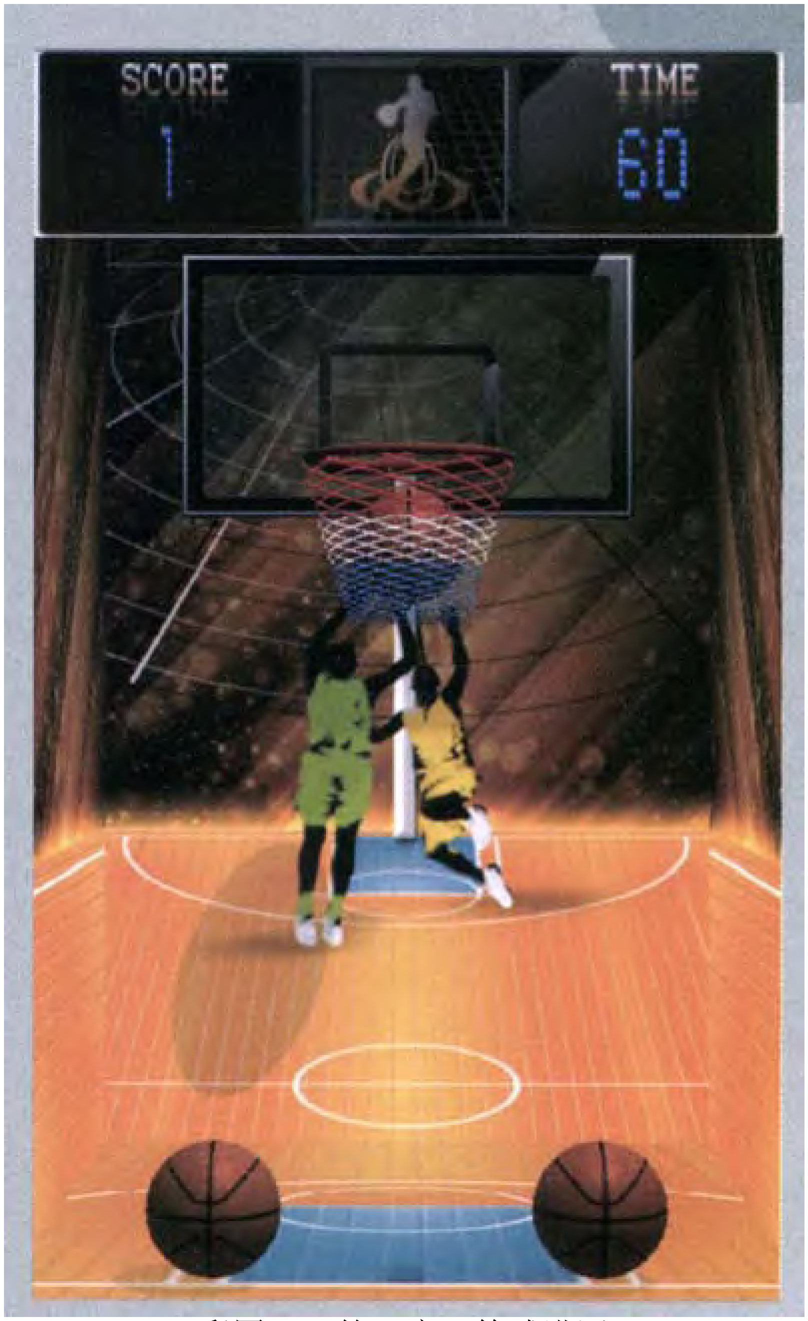


彩图34　第20章　计时模式赛艇转弯

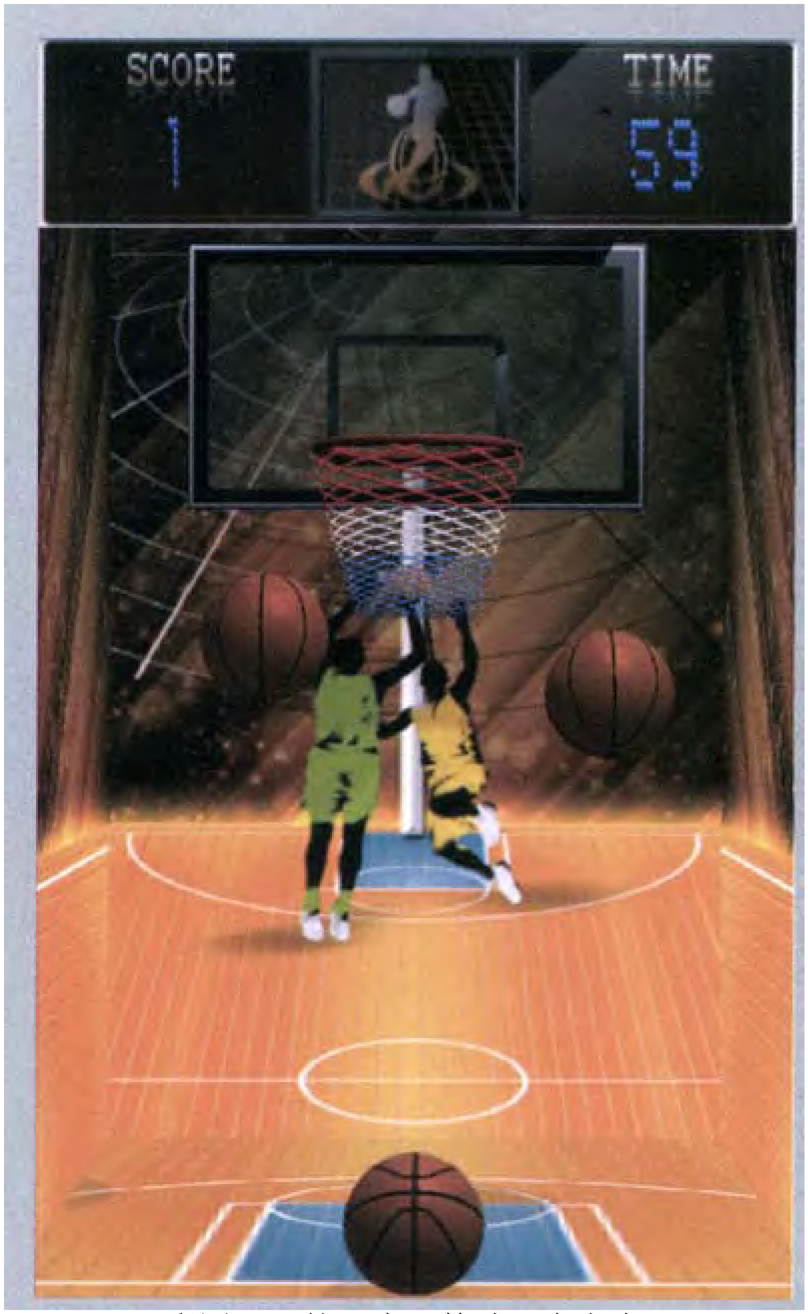
火力篮球案例开发亮点： 3D物理引擎JBullet的使用、3D拾取技术的使用、即时音效技术的使用、用顶点着色器实现软体。



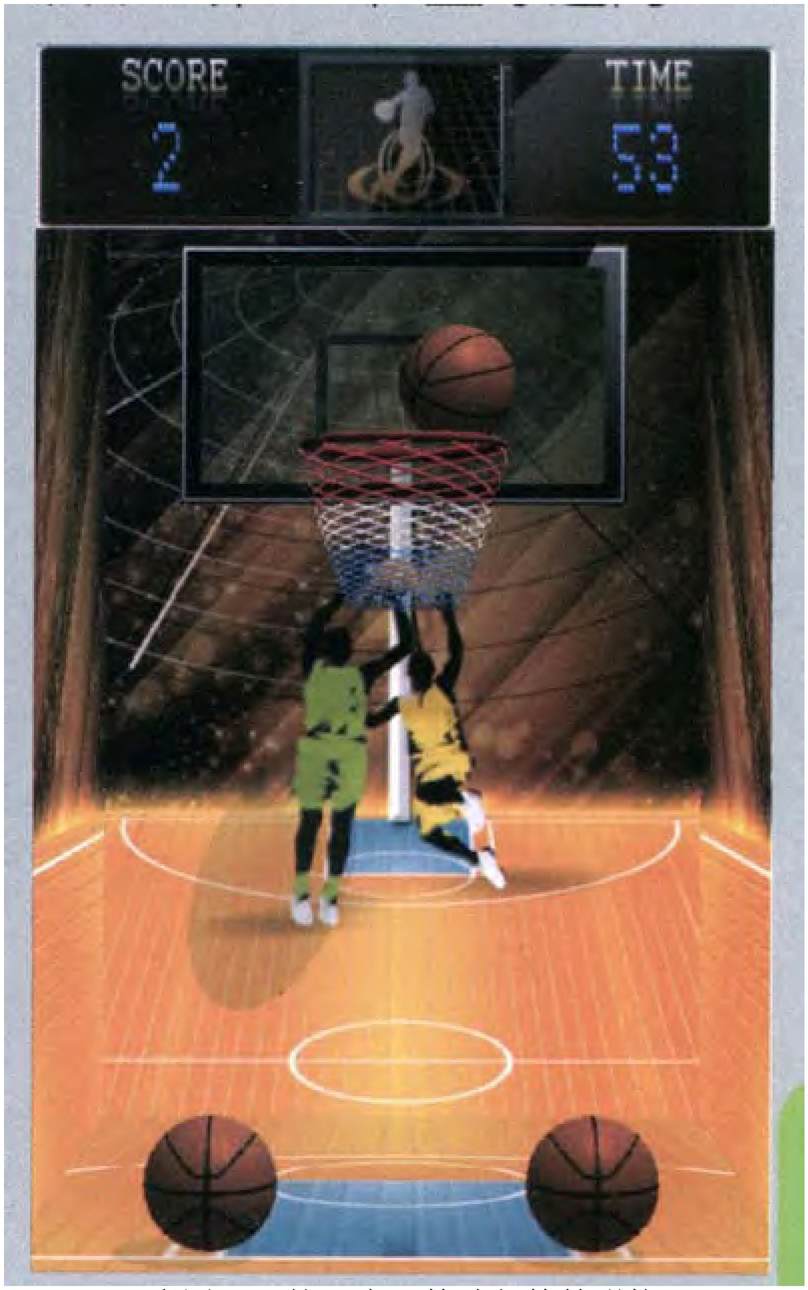
彩图35　第21章　火力篮球菜单



彩图36　第21章　篮球进网



彩图37　第21章　篮球飞在空中



彩图38　第21章　篮球与篮筐碰撞

夜鹰行动案例开发亮点： 3D模型加载技术的使用、大场景地形的制作、程序纹理与多重纹理的使用、多点触控技术的使用、传感器体感操控技术的使用。



彩图39　第22章　菜单界面



彩图40　第22章　飞机选择



彩图41　第22章　特别行动沙漠风暴



彩图42　第22章　特别行动斩首行动

Android

iPhone
Windows Phone
MeeGO PSP
PlayStation

Android 3D 游戏开发技术宝典
——OpenGL ES 2.0

吴亚峰 著　　百纳科技 审校

人民邮电出版社
北京

图书在版编目（CIP）数据

Android 3D游戏开发技术宝典：OpenGL ES 2.0/吴亚峰著．—北京：人民邮电出版社，2012.5
ISBN 978-7-115-27770-1

I．①A…　II．①吴…　III．①移动电话机—游戏程序—程序设计　IV．①TN929.53②TP311.5

中国版本图书馆CIP数据核字（2012）第041760号

内 容 提 要

随着智能手机移动嵌入式平台硬件性能的不断提升，3D游戏应用也逐渐普及开来。本书结合作者多年从事3D游戏应用开发的宝贵经验，全面介绍了与Android平台相关的必知必会的基础知识及大型完整3D案例，讲解上由浅入深，循序渐进，起点低、终点高，既适合初学者学习，也适合有一定基础的读者进一步提升之用。另外，由于OpenGL ES 2.0的着色语言通用于各种移动嵌入式平台，因此，本书中与着色器开发相关的60%左右的内容还可供iPhone、Windows Mobile、MeeGoo等平台的开发人员参考。

全书共22章，其中第1章与第2章为Android平台相关的一些基础知识；第3章～第10章介绍了基于OpenGL ES 2.0进行3D应用开发的一些必知必会的基本知识；第11章～第15章介绍了一些高级特效的实现方法；第16章～第17章介绍了3D游戏开发中相关的一些物理、碰撞检测知识以及常用的3D物理引擎JBullet；第19章介绍了3种人机交互的高级技术；第20章～第22章给出了3个完整的大型3D游戏案例，总代码量接近6万行。同时为了便于读者的学习，本书附赠的光盘中包含了书中所有案例的完整源代码，同时给出了最后3个完整大型3D游戏案例的讲解视频，最大限度地帮助读者快速掌握相应的开发技术。

本书适合Android程序员、游戏开发者及Android爱好者学习，也可以作为相关培训学校和大专院校相关专业的教学用书。

Android 3D游戏开发技术宝典——OpenGL ES 2.0

◆　著　　　　吴亚峰

审　　校　百纳科技
责任编辑　张　涛
◆　人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn

北京艺辉印刷有限公司印刷

◆　开本：800×1000　1/16
印张：47.75
字数：1419千字
印数：1-4000册
彩插：6
2012年5月第1版
2012年5月北京第1次印刷
ISBN 978-7-115-27770-1

定价：108.00元
读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第0021号
CD Link: http://pan.baidu.com/share/link?shareid=200895437&uk=1077227309&third=15
密码：ds6h

本书作者

吴亚峰，毕业于北京邮电大学，后留学澳大利亚卧龙岗大学取得硕士学位。1998年开始从事Java应用的开发，有10多年的Java开发与培训经验。主要的研究方向为OpenGL ES、手机游戏、Java EE以及搜索引擎。同时为手机游戏、Java EE独立软件开发工程师，并兼任百纳科技Java培训中心首席培训师。近十年来为数十家著名企业培养了上千名高级软件开发人员，曾编写过《Java SE 6.0编程指南》、《菜鸟成长之路——Java程序员职场全攻略》、《Android 2.0游戏开发实战宝典》、《Android核心技术与实例详解》、《Android应用案例开发大全》、《Android游戏开发大全》、《Android应用开发完全自学手册——核心技术、传感器、2D/3D、多媒体与典型案例》等多本畅销技术书籍。2008年年初开始关注Android平台下的应用开发，并开发出一系列优秀的Android应用程序与游戏。

本书在编写过程中得到了唐山百纳科技有限公司Java培训中心的大力支持，同时仇磊、李腾飞、夏学良、王旅波、李胜杰、戚悦、章雅卓、付鹏、白冰、关羽轩、郑培阳以及家人为本书的编写提供了很多帮助，在此表示衷心地感谢！

由于笔者的水平和学识有限，且书中涉及的知识较多，难免有错误疏漏之处，敬请广大读者批评指正，并多提宝贵意见，反馈邮箱javase6_guide@yahoo.cn。本书责任编辑联系邮箱为：zhangtao@ptpress.com.cn。

编者

目 录

前　　言
第1章　新一代的王者——Android概览

1.1　智能手机市场现状

1.1.1　五大智能手机操作系统
1.1.2　智能手机市场的新星

1.2　Android平台的特点及未来的趋势

1.2.1　全新理念带来的体验风暴
1.2.2　中国手机市场的主导性作用
1.2.3　手机3D游戏和应用增长迅速

1.3　如何搭建Android开发环境

1.3.1　SDK的安装及环境配置
1.3.2　Eclipse集成开发环境的搭建
1.3.3　创建并启动模拟器

1.4　Hello Android应用程序的开发

1.4.1　第一个Android应用程序
1.4.2　Android应用程序的项目结构
1.4.3　Android应用程序的调试
1.4.4　实际设备的联机调试

1.5　Android应用程序运行的机制

1.5.1　应用程序的系统架构
1.5.2　应用程序框架
1.5.3　Android运行时
1.5.4　系统库
1.5.5　底层Linux内核

1.6　本章小结

第2章　游戏开发相关Android基础知识

2.1　游戏中的音效

2.1.1　游戏中的即时音效
2.1.2　即时音效的一个案例
2.1.3　背景音乐播放技术
2.1.4　简易音乐播放器的实现

2.2　简单数据的存储——Preferences

2.2.1　Preferences简介
2.2.2　Preferences实现访问时间的记录

2.3　手机自带数据库——SQLite

2.3.1　初识SQLite
2.3.2　SQLite数据库的基本操作
2.3.3　SQLite数据库的简单案例
2.3.4　使用ContentProvider组件共享数据
2.3.5　使用ContentResolver获取分享数据

2.4　文件I/O

2.4.1　访问SD卡中的文件
2.4.2　访问手机中的文件夹
2.4.3　读取assets文件夹下的内容

2.5　2D动画的开发

2.5.1　SurfaceView用法简介
2.5.2　使用SurfaceView实现2D动画

2.6　Socket网络通信

2.6.1　Socket开发基本知识
2.6.2　服务器端
2.6.3　客户端

2.7　蓝牙通信

2.7.1　蓝牙通信的基本知识
2.7.2　聊天案例概览
2.7.3　聊天案例的开发过程

2.8　本章小结

第3章　初识OpenGL ES 2.0

3.1　OpenGL ES 2.0概览

3.1.1　OpenGL ES 2.0简介
3.1.2　初识OpenGL ES 2.0应用程序

3.2　着色器与渲染管线

3.2.1　OpenGL ES 1.x的渲染管线
3.2.2　OpenGL ES 2.0的渲染管线
3.2.3　OpenGL ES中立体物体的构建

3.3　主流Android手机GPU大PK

3.3.1　手机GPU四大家族对比
3.3.2　主流GPU的性能参数比较

3.4　本章小结

第4章　着色语言Shading Language

4.1　着色语言概述
4.2　着色语言基础

4.2.1　数据类型概述
4.2.2　数据类型的基本使用
4.2.3　运算符
4.2.4　类型转换
4.2.5　限定符
4.2.6　流程控制
4.2.7　函数的声明与使用
4.2.8　片元着色器中浮点变量精度的指定
4.2.9　程序的基本结构

4.3　特殊的内建变量

4.3.1　顶点着色器中的内建变量
4.3.2　片元着色器中的内建变量

4.4　着色语言的内置函数

4.4.1　角度转换与三角函数
4.4.2　指数函数
4.4.3　常见函数
4.4.4　几何函数
4.4.5　矩阵函数
4.4.6　向量关系函数
4.4.7　纹理采样函数
4.4.8　微分函数

4.5　本章小结

第5章　投影及各种变换

5.1　摄像机的设置
5.2　两种投影方式

5.2.1　正交投影
5.2.2　透视投影

5.3　各种变换

5.3.1　基本变换的相关数学知识
5.3.2　平移变换
5.3.3　旋转变换
5.3.4　缩放变换
5.3.5　基本变换的实质

5.4　绘制方式

5.4.1　各种绘制方式概览
5.4.2　点与线段绘制方式
5.4.3　三角形条带与扇面绘制方式
5.4.4　顶点法与索引法

5.5　设置合理的视角
5.6　卷绕和背面剪裁

5.6.1　基本知识
5.6.2　一个简单的案例

5.7　本章小结

第6章　光照

6.1　曲面物体的构建

6.1.1　球体构建的基本原理
6.2.2　案例效果概览
6.2.3　开发步骤

6.2　基本光照效果

6.2.1　光照的基本模型
6.2.2　环境光

6.2.3　散射光
6.2.4　镜面光
6.2.5　三种光照通道的合成

6.3　定位光与定向光
6.4　点法向量和面法向量
6.5　光照的每顶点计算与每片元计算
6.6　本章小结

第7章　纹理映射

7.1　初识纹理映射

7.1.1　基本原理
7.1.2　纹理映射的简单案例

7.2　纹理拉伸

7.2.1　两种拉伸方式概览
7.2.2　不同拉伸方式的案例

7.3　纹理采样

7.3.1　纹理采样概述
7.3.2　最近点采样
7.3.3　线性纹理采样
7.3.4　MIN与MAG采样
7.3.5　不同纹理采样方式的案例

7.4　mipmap纹理技术
7.5　多重纹理与过程纹理

7.5.1　案例概览
7.5.2　将2D纹理映射到球面上的策略
7.5.3　案例的场景结构
7.5.4　开发过程

7.6　本章小结

第8章　3D基本形状的构建

8.1　圆柱体

8.1.1　顶点原始位置的生成
8.1.2　案例的开发

8.2　圆锥体

8.2.1　顶点原始位置的生成
8.2.2　案例的开发

8.3　圆环体

8.3.1　顶点原始位置的生成
8.3.2　案例的开发

8.4　螺旋管

8.4.1　顶点原始位置的生成
8.4.2　案例的开发

8.5　几何球

8.5.1　顶点原始位置的生成
8.5.2　案例的开发

8.6　足球碳分子模型的搭建

8.6.1　搭建的基本原理
8.6.2　案例的开发

8.7　贝塞尔曲线及旋转面

8.7.1　三维旋转曲面的生成
8.7.2　贝塞尔曲线

8.7.3　Bezier曲线生成工具
8.7.4　印度古典建筑场景的开发

8.8　本章小结

第9章　3D模型加载

9.1　obj模型文件概览

9.1.1　obj文件的格式
9.1.2　用3DMax设计3D模型

9.2　加载obj文件

9.2.1　加载仅有顶点坐标与面数据的obj文件
9.2.2　加载后自动计算面法向量
9.2.3　加载后自动计算平均法向量
9.2.4　加载纹理坐标

9.3　本章小结

第10章　混合与雾

10.1　混合技术

10.1.1　混合基本知识
10.1.2　源因子和目标因子
10.1.3　简单混合效果案例

10.2　地月系云层效果的实现
10.3　雾

10.3.1　雾的原理与优势
10.3.2　雾的简单实现

10.4　本章小结

第11章　常用3D开发技巧

11.1　标志板

11.1.1　案例效果与基本原理
11.1.2　开发步骤

11.2　灰度图地形

11.2.1　基本原理
11.2.2　普通灰度图地形
11.2.3　过程纹理地形
11.2.4　MipMap地形

11.3　天空盒与天空穹

11.3.1　天空盒
11.3.2　天空穹
11.3.3　天空盒与天空穹的使用技巧

11.4　镜像技术

11.4.1　镜像基本原理
11.4.2　基本效果案例
11.4.3　升级效果案例

11.5　动态文本输出

11.5.1　案例效果与基本原理
11.5.2　具体开发步骤

11.6　非真实感绘制

11.6.1　案例效果与基本原理
11.6.2　具体开发步骤

11.7　本章小结

第12章　几种剪裁与测试

12.1　剪裁测试

12.1.1　基本原理与核心代码
12.1.2　一个主次视角的简单案例

12.2　Alpha测试

12.2.1　Alpha测试基本原理
12.2.2　一个椭圆窗口的案例

12.3　模板测试

12.3.1　基本原理
12.3.2　一个简单的案例

12.4　任意剪裁平面

12.4.1　基本原理
12.4.2　茶壶被任意平面剪裁的案例

12.5　本章小结

第13章　顶点着色器的妙用

13.1　飘扬的旗帜

13.1.1　基本原理
13.1.2　开发步骤

13.2　扭动的软糖

13.2.1　基本原理
13.2.2　开发步骤

13.3　风吹椰林场景的开发

13.3.1　椰子树随风摇摆的基本原理
13.3.2　开发步骤

13.4　展翅飞翔的雄鹰

13.4.1　基本原理
13.4.2　开发步骤

13.5　二维扭曲

13.5.1　基本原理
13.5.2　开发步骤

13.6　OpenGL ES 1.x与OpenGL ES 2.0实现方案的对比
13.7　本章小结

第14章　片元着色器的妙用

14.1　程序纹理技术

14.1.1　砖块着色器
14.1.2　沙滩球着色器

14.2　数字图像处理

14.2.1　卷积的基本知识
14.2.2　平滑过滤
14.2.3　边缘检测
14.2.4　锐化处理
14.2.5　浮雕效果
14.2.6　图像渐变

14.3　分形着色器

14.3.1　曼德布罗集简介
14.3.2　曼德布罗集着色器的实现
14.3.3　将曼德布罗集纹理应用到实际物体上
14.3.4　茱莉亚集着色器的实现

14.4　本章小结

第15章　真实光学环境的模拟

15.1　投影贴图

15.1.1　案例效果与基本原理
15.1.2　开发步骤

15.2　反射环境模拟

15.2.1　案例效果与基本原理
15.2.2　开发步骤

15.3　折射环境模拟

15.3.1　案例效果与基本原理
15.3.2　开发步骤

15.4　色散效果的模拟

15.4.1　案例效果与基本原理
15.4.2　开发步骤

15.5　菲涅尔效果的模拟

15.5.1　案例效果与基本原理
15.5.2　开发步骤

15.6　凹凸映射

15.6.1　案例效果与基本原理
15.6.2　法向量纹理图的生成
15.6.3　案例的开发

15.7　平面阴影

15.7.1　案例效果与基本原理
15.7.2　开发步骤

15.8　阴影映射

15.8.1　案例效果与基本原理
15.8.2　距离纹理图的生成
15.8.3　阴影场景的绘制

15.9　光线跟踪

15.9.1　案例效果与基本原理
15.9.2　开发步骤

15.10　本章小结

第16章　游戏开发中的物理学

16.1　碰撞检测基本技术

16.1.1　AABB包围盒的基本原理
16.1.2　AABB包围盒的计算
16.1.3　AABB包围盒的碰撞检测
16.1.4　一个AABB包围盒的案例
16.1.5　旋转后的AABB包围盒
16.1.6　AABB包围盒的使用要点
16.1.7　OBB包围盒

16.2　穿透效应

16.2.1　案例的运行效果与基本原理
16.2.2　具体开发步骤

16.3　粒子系统

16.3.1　案例运行效果与基本原理
16.3.2　CPU版案例的开发
16.3.3　GPU版案例开发步骤

16.4　弹簧质点模型

16.4.1　案例运行效果与基本原理
16.4.2　具体开发步骤

16.5　本章小结

第17章　游戏的心脏——物理引擎

17.1　物理引擎很重要

17.1.1　什么是物理引擎
17.1.2　常见的物理引擎

17.2　JBullet物理引擎

17.2.1　基本的物理学概念
17.2.2　JBullet中常用类的介绍

17.3　箱子相互碰撞的案例

17.3.1　案例运行效果及准备工作
17.3.2　案例的基本框架结构
17.3.3　常量类——Constant
17.3.4　3D场景渲染类——MySurfaceView
17.3.5　水平地面——TexFloor类
17.3.6　箱子——TexCube类

17.4　复合碰撞形状的使用

17.4.1　案例运行效果
17.4.2　立方体圆柱复合形状——CubeCylinder类

17.5　凹凸地形的案例

17.5.1　案例运行效果
17.5.2　地形类——LandForm

17.6　任意形状物体的碰撞

17.6.1　案例运行效果
17.6.2　加载物体类——LoadedObjectVertexNormal
17.6.3　加载物体刚体类——LoadRigidBody

17.7　铰链关节

17.7.1　铰链关节的基本知识
17.7.2　案例的运行效果
17.7.3　铰链关节旋转角速度的计算
17.7.4　3D界面渲染类——MySurfaceView

17.8　滑动关节

17.8.1　滑动关节的基本知识
17.8.2　案例效果图
17.8.3　3D界面渲染类——MySurfaceView

17.9　六自由度关节

17.9.1　六自由度关节的基本知识
17.9.2　案例的运行效果
17.9.3　人偶类——Doll
17.9.4　拾取时采用的点对点关节

17.10　本章小结

第18章　传感器应用的开发

18.1　基本的开发流程
18.2　加速度传感器

18.2.1　加速度传感器简介
18.2.2　案例的开发

18.3　磁场传感器

18.3.1　磁场传感器简介
18.3.2　案例的开发

18.4　光传感器

18.4.1　光传感器简介
18.4.2　案例的开发

18.5　温度传感器

18.5.1　温度传感器简介
18.5.2　案例的开发

18.6　接近传感器

18.6.1　接近传感器简介
18.6.2　案例的开发

18.7　姿态传感器

18.7.1　姿态传感器简介
18.7.2　案例的开发

18.8　本章小结

第19章　游戏开发小贴士

19.1　3D拾取技术

19.1.1　案例效果与基本原理
19.1.2　开发步骤

19.2　多点触控

19.2.1　案例效果与基本原理
19.2.2　开发步骤

19.3　多键监听

19.3.1　案例效果与基本原理
19.3.2　开发步骤

19.4　本章小节

第20章　BN赛艇

20.1　游戏背景及功能概述

20.1.1　背景概述
20.1.2　功能介绍

20.2　游戏的策划及准备工作

20.2.1　游戏的策划
20.2.2　Android平台下游戏的准备工作

20.3　游戏的架构

20.3.1　各个类简要介绍
20.3.2　游戏框架简介

20.4　主控制类MyActivity
20.5　2D界面相关类

20.5.1　欢迎界面类WelcomeView
20.5.2　2D界面父类MySFView
20.5.3　主菜单类MenuView
20.5.4　2D界面绘制类ViewForDraw
20.5.5　数据库工具类DBUtil
20.5.6　Android系统版本对话框AndroidVersionDialog

20.6　选船界面相关类

20.6.1　着色器管理类ShaderManager
20.6.2　围墙类ColorLightRect
20.6.3　选船房间类HouseForDraw
20.6.4　展台类DisplayStation
20.6.5　赛艇类Boat
20.6.6　选船界面XCSurfaceView

20.7 游戏界面相关类

20.7.1 进度条类Process
20.7.2 3D物体父类BNDrawer
20.7.3 3D物体控制类TDObjectForControl
20.7.4 赛艇类Boat
20.7.5 可碰撞物体父类KZBJDrawer
20.7.6 可碰撞物体控制类KZBJForControl
20.7.7 赛艇与可碰撞物体检测碰撞线程ThreadColl
20.7.8 可吃物体父类SpeedForEat
20.7.9 可吃物体控制类SpeedForControl
20.7.10 晶体旋转线程RotateThread
20.7.11 赛艇与晶体碰撞检测线程ThreadForEat
20.7.12 比较两灌木距离摄像机距离类MyComparable
20.7.13 单个灌木类SingleShrub
20.7.14 灌木控制类ShrubForControl
20.7.15 赛艇赛道类RaceTrack
20.7.16 赛艇与桥碰撞检测类PZZ
20.7.17 赛艇运动路线工具类PathUtil
20.7.18 尾浪类WeiLang
20.7.19 水面类Water
20.7.20 游戏界面类MyGLSurfaceView
20.7.21 运动监听线程KeyThread

20.8 该游戏中的着色器
20.9 游戏的优化与改进

第21章 火力篮球

21.1 游戏背景及功能概述

21.1.1 背景概述
21.1.2 功能介绍

21.2 游戏的策划及准备工作

21.2.1 游戏的策划

21.2.2　Android平台下游戏开发的准备工作

21.3　游戏的架构

21.3.1　各个类简要介绍
21.3.2　游戏框架简介

21.4　公共类BasketBall_Shot_Activity
21.5　辅助界面相关类

21.5.1　主菜单界面CaiDanView类
21.5.2　设置界面ShengyinKGJiemian类
21.5.3　关于界面GuanYuView类
21.5.4　记录界面JiLuView类
21.5.5　数据库相关类SQLiteUtil

21.6　游戏界面相关类

21.6.1　游戏界面绘制类GLGameView
21.6.2　着色器程序管理器类ShaderManager
21.6.3　添加刚体进物理世界类TianjiaBody
21.6.4　添加胶囊刚体进物理世界类JiaoNangTianjiaBody
21.6.5　篮球控制类BasketBallForDraw
21.6.6　纹理矩形绘制类WenLiJuXing

20.7　游戏中着色器的开发

21.7.1　纹理的着色器
21.7.2　光照着色器
21.7.3　影子着色器
21.7.4　仪表盘背景着色器
21.7.5　篮网着色器

21.8　游戏的优化与改进

第22章　夜鹰行动

22.1　游戏背景及功能概述

22.1.1　背景概述
22.1.2　功能介绍

22.2　游戏的策划及准备工作

22.2.1　游戏的策划
22.2.2　Android平台下游戏开发的准备工作

22.3　游戏的架构

22.3.1　各个类的简要介绍
22.3.2　游戏框架简介

22.4　公共类Aircraft Activity
22.5　游戏相关类

22.5.1　游戏界面绘制类GLGameView
22.5.2　游戏线程类KeyThread
22.5.3　子弹类BulletForControl
22.5.4　坦克类TanKe
22.5.5　山地绘制类LandForm
22.5.6　颜色圆面类CircleForDraw
22.5.7　星空类SkyNight
22.5.8　爆炸效果绘制类DrawBomb
22.5.9　数字绘制类NumberForDraw

22.6　该游戏中的着色器

22.6.1　加载界面着色器
22.6.2　地形着色器
22.6.3　按钮着色器
22.6.4　星空着色器
22.6.5　生命值矩形着色器
22.6.6　颜色圆面着色器

22.7　游戏的优化与改进

前　　言

为什么要写这样的一本书

随着智能手机移动嵌入式平台硬件性能的不断提升，如水果忍者、极品飞车、会说话的Tom猫等一批优秀的3D游戏娱乐应用在广大智能机用户间流行开来，与此同时，也带动了手机游戏产业逐渐从2D走向3D。但目前国内专门介绍这方面开发的书籍与资料都非常少，同时3D应用开发的门槛又比较高，使得很多初学者无从下手。根据这种情况，笔者结合多年从事3D游戏应用开发的经验编写了这样一本书。

了解一些3D领域的技术人员都知道，移动嵌入式平台上的3D应用开发主要是基于OpenGL ES的3个版本，1.0、1.1和2.0。OpenGL ES 1.0与1.1是较老的版本，渲染能力很有限，留给开发人员发挥的空间也很有限。而本书主要介绍了OpenGL ES 2.0，其采用的是可编程渲染管线，留给开发人员充分的发挥空间，可以说是“海阔凭鱼跃，天高任鸟飞”。

虽然本书中的案例都是基于Android平台开发的，但其中关于可编程渲染管线着色器的知识可以通用于大部分的移动嵌入式平台，包括iPhone、Windows Mobile、MeeGO、PSP、PlayStation等。因此，学习Android平台下高级3D应用开发的朋友可以参考此书，希望学习其他嵌入式平台下3D应用高级开发的朋友也可以使用书中60%左右的内容。

经过半年多见缝插针式的奋战，本书终于交稿了。回顾写书的这半年多时间，不禁为自己能最终完成这个耗时费力的“大制作”而感到欣慰。同时也为自己能将从事游戏开发近10年来积累的宝贵经验以及编程感悟分享给正在开发阵线上埋头苦干的广大编程人员而感到高兴。

贾岛的《剑客》一诗有言：“十年磨一剑，霜刃未曾试，今日把示君，谁有不平事？”从1998年首次接触Java与OpenGL算起，到现在也是10年有余。笔者希望用10年的知识和经验磨出的利剑能够帮助广大读者在实际工作中披荆斩棘、奋勇向前。

本书特点

1．内容丰富，由浅入深

本书组织上本着“起点低，终点高”的原则，内容覆盖了从学习OpenGL ES 2.0必知必会的基础知识到基于OpenGL ES 2.0实现高级特效，最后还给出了3个完整的大型3D游戏案例。这3个案例均是笔者带领学员参加谷歌（Google）2011第二届Android应用开发中国大学生挑战赛的获奖作品，第22章的夜鹰行动更是获全国总决赛前5名大奖的作品。

这样的内容组织使得懵懂的Android开发菜鸟可以一步一步成长为3D开发的达人，符合绝大部分想学习3D开发的学生与程序人员以及正在学习3D开发的技术人员的需求。

2．结构清晰，讲解到位

本书中配合每个需要讲解的知识点都给出了丰富的插图与完整的案例，使得初学者易于上手，有一定基础的读者便于深入。书中所有的案例均是根据笔者多年的开发心得进行设计的，结构清晰明朗，便于读者进行学习。同时书中还给出了很多笔者多年来积累的编程技巧与心得，具有很高的参考价值。

3．非常实用的光盘

为了便于读者学习，本书附赠的光盘中包含了书中所有案例的完整源代码，同时还为最后的3个完整大型3D游戏案例配置了讲解视频，最大限度地帮助读者掌握技术。

内容导读

本书总共分为22章，内容按照必知必会的基础知识、基于OpenGL ES 2.0实现高级特效以及真实大型游戏案例的顺序进行详细地讲解。

章　名	主要内容
第1章　新一代的王者——Android概览	本章简要介绍了Android的诞生、特点、开发环境的搭建以及其运行机制
第2章　游戏开发相关Android基础知识	本章主要介绍了基于Android平台开发游戏所必需了解的基础知识，包括音效、数据库、I/O以及网络通信等
第3章　初识OpenGL ES 2.0	本章介绍了OpenGL ES的一些基本知识，内容主要包括OpenGL ES 1.x与OpenGL ES 2.0的渲染管线
第4章　着色语言——Shading Language	本章对用于实现OpenGL ES 2.0可编程渲染管线着色器的着色语言进行了系统地介绍，为后面各方面的深入介绍打下了基础
第5章　投影及各种变换	本章介绍了3D开发中投影、各种变换的原理与实现以及几种不同的绘制方式
第6章　光照	本章介绍了OpenGL ES 2.0中光照的基本原理与实现、点法向量与面法向量的区别以及光照的每顶点计算与每片元计算的差别
第7章　纹理映射	本章介绍了纹理映射的基本原理与使用，同时还介绍了不同的纹理拉伸与采样方式以及多重过程纹理技术
第8章　3D基本形状的构建	本章介绍了使用OpenGL ES 2.0开发各种3D基本形状的原理与实现，包括圆柱、圆锥、圆环、螺旋管、几何球以及贝塞尔曲线旋转面等内容
第9章　3D模型加载	本章介绍了如何使用自定义的加载工具类直接加载使用3DMax创建的3D立体物体
第10章　混合与雾	本章主要介绍混合以及雾的基本原理与使用
第11章　常用3D开发技	本章主要介绍了一些常用的3D开发技巧，包括标志版、灰度图地形生成、天空盒与天空穹、镜像技术、

巧	动态文本输出以及非真实感绘制等
第12章 几种剪裁与测试	本章主要介绍了OpenGL ES中经常使用的几种剪裁与测试，包括剪裁测试、Alpha测试、模板测试以及任意剪裁平面等
第13章 顶点着色器的妙用	本章主要介绍了如何通过顶点着色器实现OpenGL ES 1.x不容易实现的酷炫效果，包括飘扬的旗帜、扭动的软糖、展翅飞翔的雄鹰等
第14章 片元着色器的妙用	本章介绍了如何通过片元着色器实现OpenGL ES 1.x不易实现的酷炫效果，包括程序纹理、数字图像处理、分形着色器等
第15章 真实光学环境的模拟	本章介绍了如何通过OpenGL ES 2.0模拟现实世界的一些常见光学效果，如投影、反射、折射、阴影等
第16章 游戏开发中的物理学	本章介绍了3D游戏开发中常用的一些物理学以及碰撞检测的知识，包括AABB包围盒、穿透效应、粒子系统以及弹簧质点模型等
第17章 游戏的心脏——物理引擎	本章介绍了3D游戏开发中经常使用的物理引擎JBullet，包括JBullet中的一些基本概念、各种形状物体的碰撞以及几种关节的使用
第18章 传感器应用的开发	本章介绍了Android手机中各种传感器的使用，包括加速度传感器、磁场传感器、光传感器、温度传感器、接近传感器以及姿态传感器等
第19章 游戏开发小贴士	本章主要介绍了游戏开发中经常使用的一些人机交互技术，包括3D拾取、多点触控以及多键监听等
第20章 BN赛艇	本章给出了一个完整的3D游戏案例——BN赛艇，内容涉及大场景的制作、赛艇与物体的碰撞、选择不同的赛艇、多艘赛艇同时竞速以及音效设置等各种功能的开发
第21章 火力篮球	本章给出了一个使用JBullet物理引擎的3D游戏案例——火力篮球，内容涉及JBulet物理引擎的使用、音效设置以及3D物体的拾取等
第22章 夜	本章给出了一款完整的3D空战游戏案例——夜鹰行

鹰行动	动，内容主要涉及大场景的制作、飞机选择、3D模型的加载、飞机与物体的碰撞、子弹发射、体感操控等功能的开发

本书内容丰富，从基本知识到高级特效；从简单的应用程序到完整的3D游戏案例，适合不同需求、不同水平层次的各类读者。

- 初学Android下OpenGL ES 2.0应用开发的读者

本书包括在Android平台下进行3D应用开发各方面的知识，内容由浅入深，配合详细的案例。非常适合3D游戏的初学者循序渐进地学习，最终成为3D游戏应用开发达人。

- 有一定3D开发基础希望进一步深入学习OpenGL ES 2.0高级3D开发技术的读者本书不仅包括了OpenGL ES 2.0开发的基础知识，同时也包括基于OpenGL ES 2.0实现高级特效以及3D物理引擎与完整的游戏案例，有利于有一定基础的开发人员进一步提高开发水平与能力。

- 其他移动嵌入式平台的3D开发人员

由于无论在哪种平台下，OpenGL ES 2.0应用的渲染核心都是使用统一的着色语言开发的，故虽然本书的案例是基于Android平台，但其中60%的内容与iPhone、Windows Mobile、MeeGo等平台是通用的。因此其他移动嵌入式平台的3D开发人员也可以参考此书进行深入学习。

第1章　新一代的王者——Android概览

随着移动互联网时代的到来，智能手机逐渐走进了人们的生活，Google公司也于2007年11月5日发布了基于Linux平台的开源手机操作系统——Android。由于Android系统的开源性以及其他各个方面的因素，其受到了广大手机厂商的青睐，因此需要大量的Android开发人员来满足日益增长的海量软件开发需求。

1.1　智能手机市场现状

手机的问世使得人们联络更加方便，同时随着手机的逐渐发展，其已经成为现代生活中不可或缺的一个组成部分。在移动互联网时代，手机也可以像PC一样安装很多的游戏、应用，就像一部随身携带的小型计算机，而且其有着PC不可比拟的移动性，这使得智能手机的销量逐渐增大。

1.1.1 五大智能手机操作系统

移动互联网时代到来之前，智能手机操作系统主要有Symbian、BlackBerry、Windows Mobile、嵌入式Linux和Palm OS。这5大操作系统占据了整个智能手机市场，并且以Symbian为主，其市场占有率最高，曾经一度达到惊人的70%。下面就对这5种智能手机操作系统进行简单的介绍。

- Symbian

Symbian操作系统的前身是EPOC操作系统。1998年6月，由诺基亚、摩托罗拉、爱立信、三菱和宝意昂5家公司在英国伦敦共同投资创建了Symbian公司，2008年被诺基亚全额收购。诺基亚公司曾推出了一系列的基于Symbian操作系统的智能手机，其中较为知名的机型有Nokia 5800等，如图1-1所示。

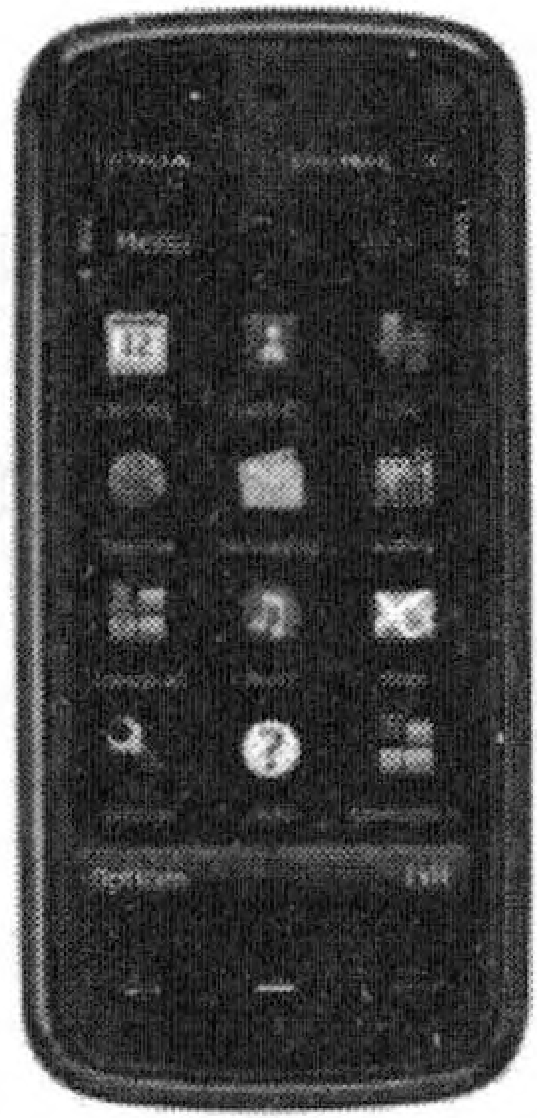


图1-1　Nokia 5800

- BlackBerry

BlackBerry是加拿大RIM公司推出的一款移动通话设备，其支持电子邮件、电话、文字短信、网页浏览及其他无线资讯服务，曾被用在许多会议或者谈判中，基本机型如图1-2所示。



图1-2　BlackBerry

- Windows Mobile

Windows Mobile是微软为手持设备推出的移动版“Windows”，该平台的桌面不仅与Windows类似，同时内核以及使用方式也与Windows大致相同，基于该平台的手机对CPU和内存都有较高的要求。多普达T8388就是一款基于Window Mobile平台的手机，如图1-3所示。



图1-3　多普达T8388

- 嵌入式Linux

嵌入式Linux是Linux在嵌入式领域的一个变体，与Linux一样，也是一款开源的操作系统，目前有许多版本。自从Linux进入嵌入式终端领域后，许多移动终端厂商就被其开源特性吸引，主要的厂商有摩托罗拉以及NTT DoCoMo等，其中摩托罗拉的A1200堪称经典，如图1-4所示。但是因为其很多方面不符合手机终端用户的使用习惯，所以一直以来未得到业界的广泛支持，成功的机型不多。



图1-4　MOTO A1200

- Palm OS

Palm OS是Palm公司开发的一款专门用于PDA（掌上电脑）的操作系统，该系统一度是PDA上的霸主，曾经占据近90%的PDA市场份额。虽然该系统并不是专门针对于手机设计的，但是该系统的优良特性和对移动设备良好的支持使得其同样可以成为优秀的手机操作系统，如图1-5所示。



图1-5　Palm手机

1.1.2　智能手机市场的新星

2007年1月9日，iPhone的诞生开创了移动设备的新纪元，其对移动电话做出了一个全新的阐释。无线通信、媒体播放器、数码助理、多点触控技术、传感器体感以及数以万计的软件应用，苹果引领着全世界移动设备的潮流，神人乔布斯再次创造了神话。iPhone的基本外观如图1-6所示。



图1-6　iPhone手机

同年，11月5日Google发布了一款基于Linux平台的开源手机操作系统Android，该平台由 操作系统、用户界面和应用程序组成，是首个为移动终端打造的具有开放性质的平台。早期的设备中，比较具有代表性的是摩托罗拉公司的XT702（里程碑1），如图1-7所示。



图1-7　XT702手机

1.2　Android平台的特点及未来的趋势

Android平台基于Linux，其不仅继承了Linux的各种优良特性，同时由于“Android之父”Andy Rubin的先进理念，使得Android平台又增加了许多非常具有吸引力的新特性。这不仅给终端用户带来了非凡的体验，还大大减轻了Android应用开发者的工作量与难度。

1.2.1　全新理念带来的体验风暴

Android是具有开源性质的手机终端解决方案，开发人员可以自由地获取以及修改相应的源代码，具体基本特性如下。

- 开放性

Android平台同时包括底层的操作系统、上层的用户界面和应用程序，这些全部是对开发者及厂商开放的。

- 应用程序平等

Android系统中任何的应用程序均是平等的，不存在传统智能手机操作系统中系统软件与应用软件的区别与障碍。即用户可以随意地替换Android系统中原有的一些应用程序，如拨号软件、短信软件以及照相机等。

- 应用程序无界限

Android平台打破了应用程序之间的界限，一款应用程序不但可以通过API访问系统提供的功能，还可以声明自身的功能供其他应用程序调用。或在其他应用程序允许的情况下调用其他应用程序的组件完成自身的功能，这在以往的智能手机操作系统中是不可想象的。

- 快速方便的应用程序开发

Android平台为开发人员提供了丰富的库和多样的工具，这使得开发人员可以快速、方便地开 发各种不同功能的应用程序。同时由于Android平台内置了Google地图，使得LBS等位置相关的应用程序开发也变得极其简便。

1.2.2　中国手机市场的主导性作用

2011年5月20日，美国移动广告网络公司（Millennial Media）公布了各个智能手机操作系统平台的新增市场占有率，谷歌Android操作系统继续在全球智能手机市场上占据主导地位，所占份额为52%；苹果iOS排名第二，所占份额为28%；RIM黑莓操作系统第三，份额为16%；Symbian及其他系统所占总份额为4%。

Android相对于iPhone具有更高的性价比，其更加适合国内大部分用户购买并使用。并且随着Android的逐渐成熟，许多其他系统中，如有的著名游戏及应用均发布了Android版本，这大大增加了Android的软件资源，进一步增强了Android的用户凝聚力。

“得中国市场者得天下”，中国是目前世界上人口最多、经济发展最迅猛的国家，中国国内Android市场目前就占到了全世界的20%，以后可能还会进一步增加。相信Android在夺下中国市场之后，可以稳坐智能手机霸主的宝座。

1.2.3　手机3D游戏和应用增长迅速

随着移动互联网和后PC时代的到来，移动手持设备的性能大幅提升，手机的功能也越来越多，越来越强大，这使得2D手机软件不再满足以追求酷炫为目标的移动手持设备。

如今的手机游戏也远远不再是传统印象中的“俄罗斯方块”、“贪吃蛇”之类画面简陋，规则简单的游戏，已经发展到了具有很强的娱乐性和交互性的复杂形态了。

人们在上班之余、上下班途中，玩会儿3D手机游戏已经成为了一种潮流，这在北京、上海和广东的地铁、公交车上随处可见。如图1-8和图1-9所示，都是最近很流行的3D Android手机游戏。

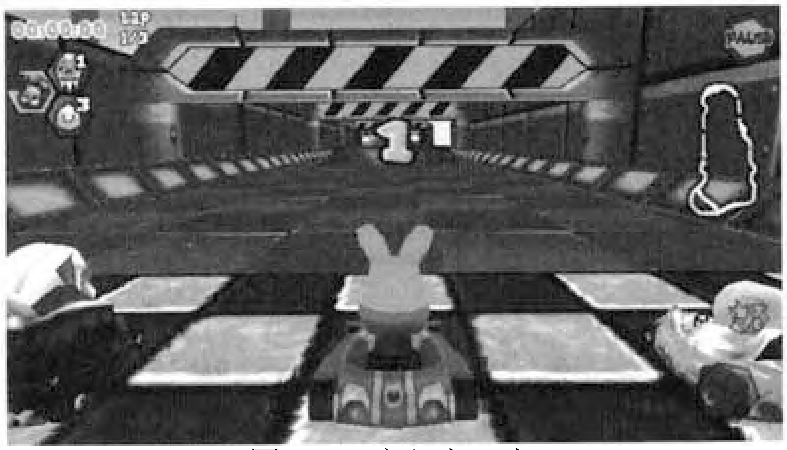


图1-8　疯狂卡丁车

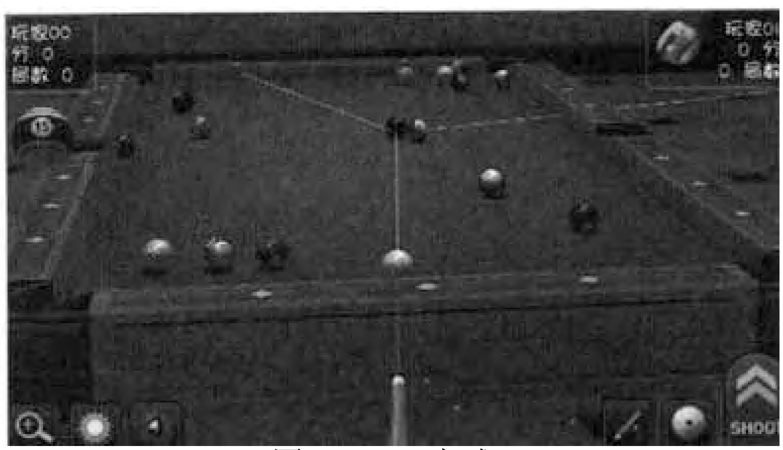


图1-9　3D桌球

同时，据美国的一家咨询公司的数据显示，从2010年开始，3D应用程序的下载量就稳步增长，增长速度远远超过2D的游戏娱乐应用，这个趋势使得手机3D游戏和应用在智能手机软件中所占的比例日益增加，3D必然是未来的王者。

1.3　如何搭建Android开发环境

Eclipse是一款开放源代码、基于Java的可扩展开发平台。其包括一个框架和一组服务，主要 通过插件来构建开发环境。而Android的开发工具即为Eclipse的插件，这也减少了开发人员的开发成本。本节主要讲解基于Eclipse的Android开发环境的搭建、模拟器的创建和运行等相关知识。

1.3.1 SDK的安装及环境配置

本小节主要介绍Android开发中相关SDK的安装及环境配置，主要包括JDK和Android SDK，具体步骤如下。

- 首先登录Oracle网站 http://www.oracle.com/technetwork/java/javase/downloads/index.html，下载最新的适合自己开发PC或工作站操作系统版本的JDK安装程序。单击如图1-10所示的按钮进入如图1-11所示下载页面。

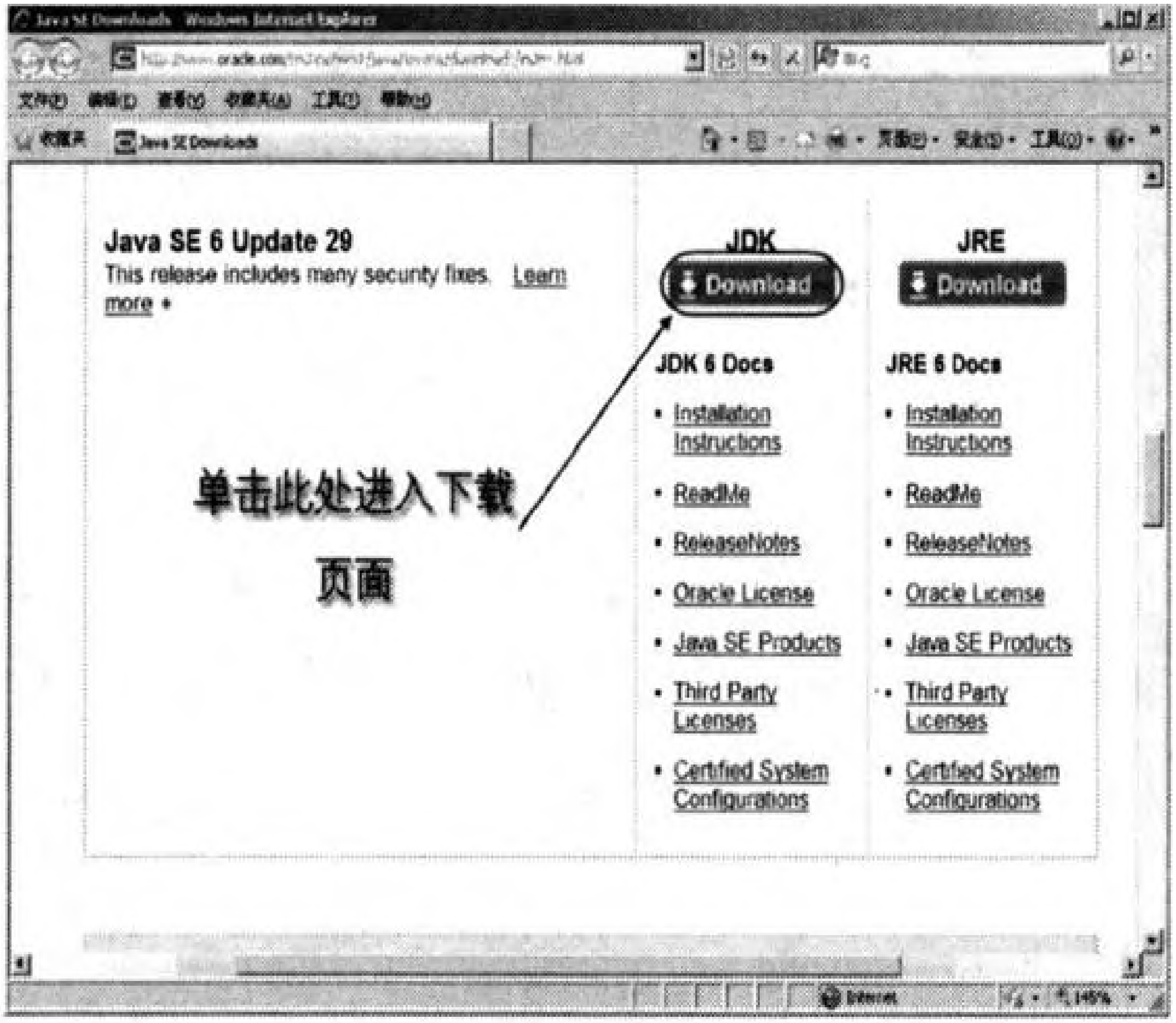


图1-10　JDK下载页面

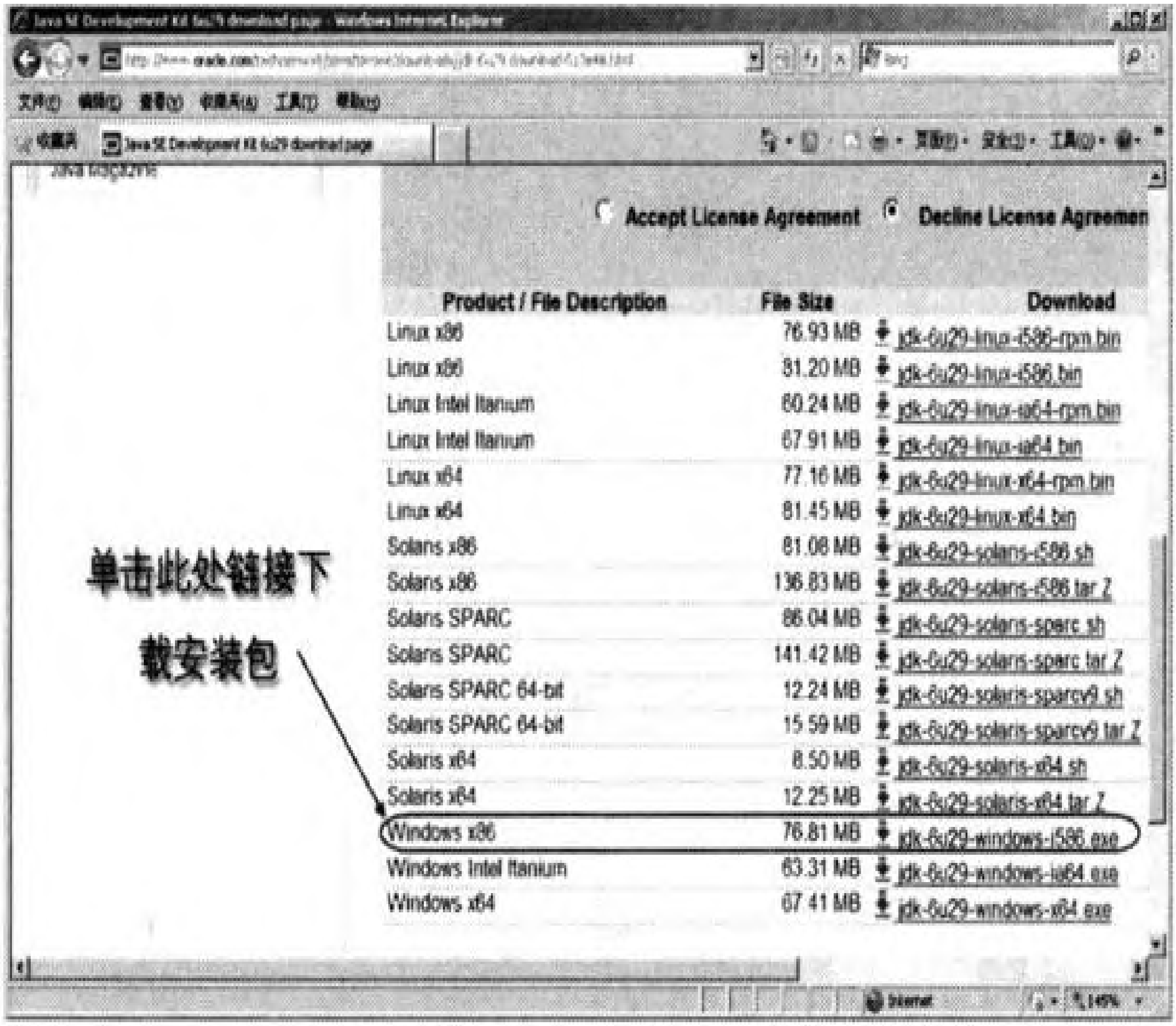


图1-11　JDK下载页面

提示

请读者注意选择下载合适的版本，32位操作系统应下载X86版本；64位的应下载X64版本。

- 接着双击下载的JDK安装包，开始JDK的安装。安装过程中，系统会弹出如图1-12所示的安装设置界面，若没有特殊需要，单击下一步按钮安装到默认目录即可。当然，也可以单击“更改”按钮设置JDK的安装路径。

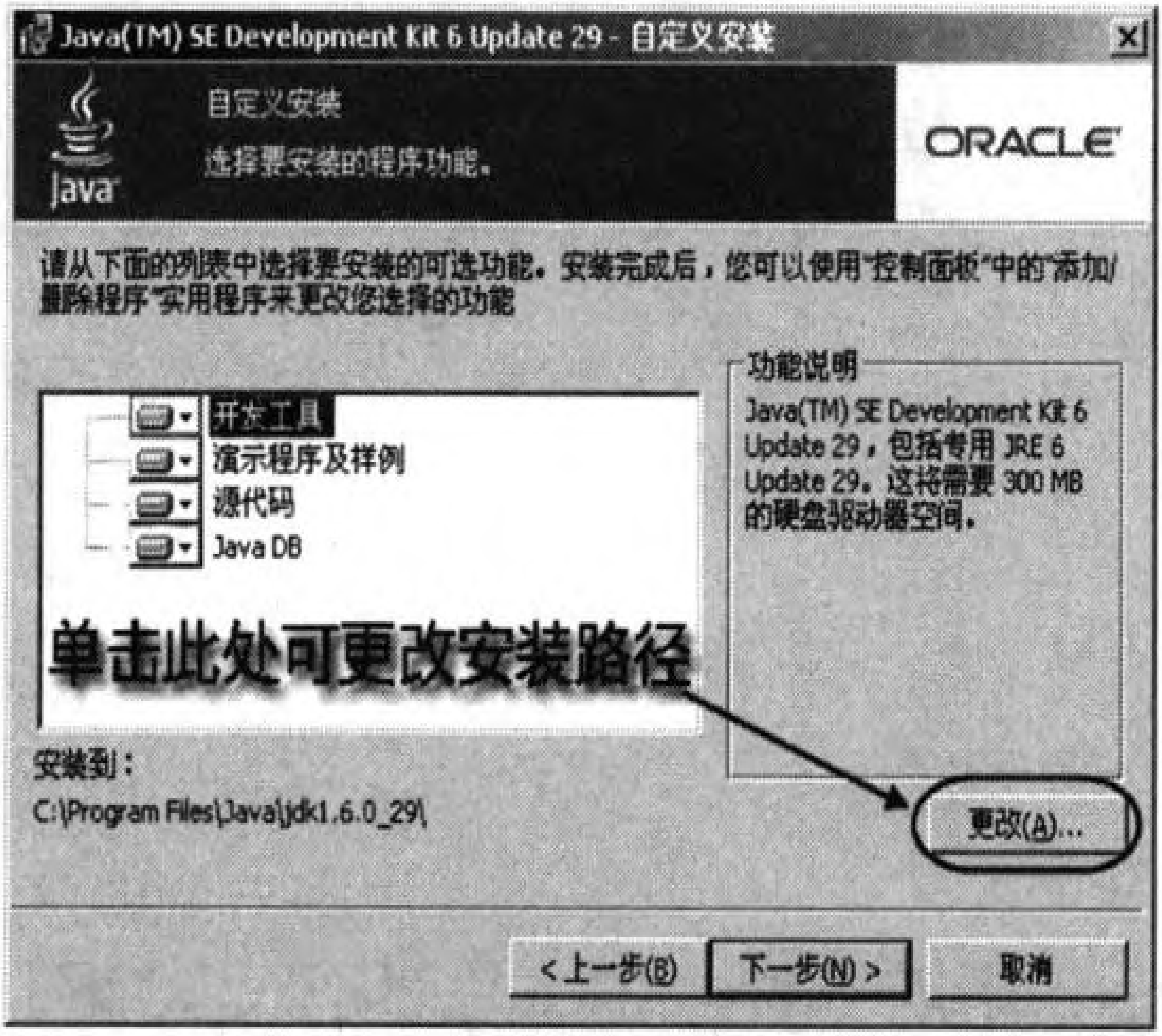


图1-12　JDK安装界面

- 安装完成后将转到如图1-13所示界面，单击“完成”按钮结束安装。



图1-13　安装完成

- 接着需要在操作系统的Path环境变量中加入JDK的bin路径，用鼠标右键单击“我的电脑”图标，选择属性→高级→环境变量，如图1-14所示。在Path变量中添加JDK的bin路径，如C:\Program Files\Java\jdk1.6.0_29\bin，并且与前面原有的环境变量用“；”分开。

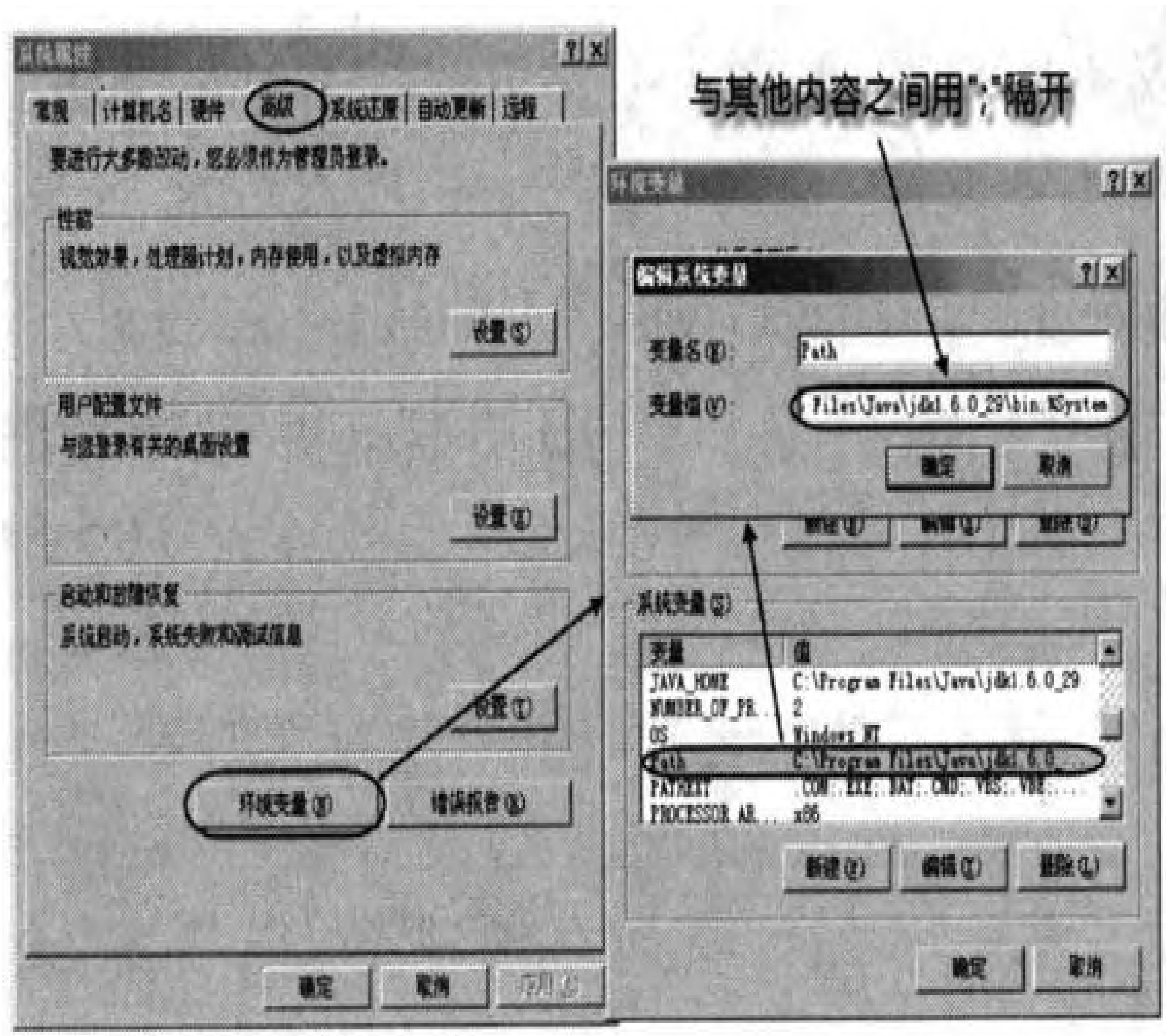


图1-14　设置JDK环境变量

• 然后在环境变量中新增JAVA_HOME项。具体方法为，在环境变量下的系统变量中添加JAVA_HOME项，将变量值设置为JDK的安装路径，如C:\Program Files\Java\jdk1.6.0_29。整个操作过程如图1-15所示。

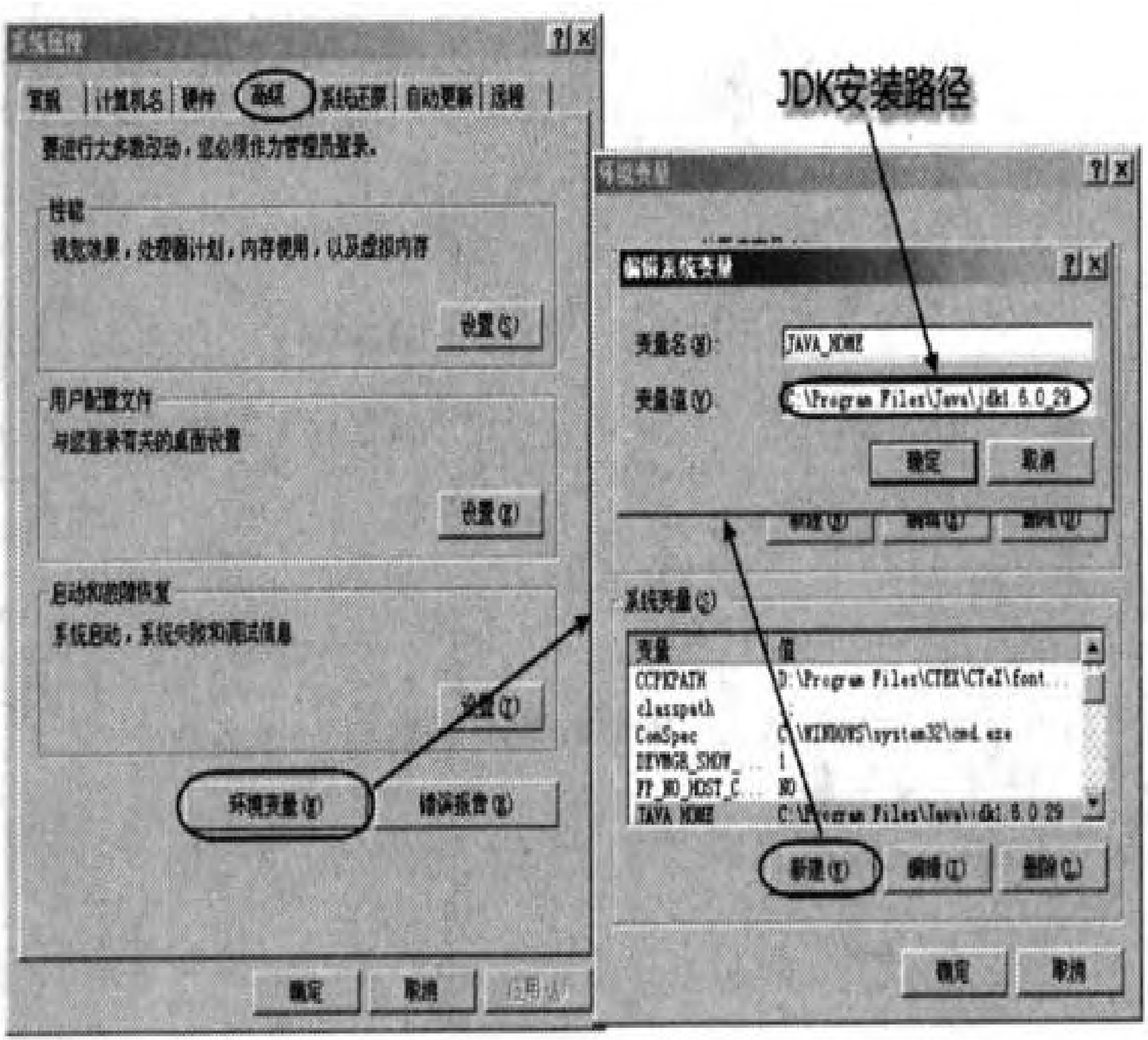


图1-15　新建JAVA_HOME项

- 接着登录http://developer.android.com页面，下载Android SDK的安装压缩包，如图1-16所示。



图1-16　SDK下载页面

- 选择图1-16页面中的SDK进入如图1-17所示的下载页面，选择合适版本的Android SDK安装压缩包进行下载。



图1-17　SDK下载页面

> **提示**
>
> 本书中介绍的是下载使用2.3版。读者也可以自行下载其他版本，方法与之类似。

• 将下载完毕的Android SDK压缩包解压到硬盘某个目录下。

• 与前面介绍的将JDK的bin路径添加到Path环境变量中的方法相同，将解压后android-sdk-windows文件夹下的tools目录路径添加到Path环境变量中，然后单击“确定”按钮完 成所有环境变量的配置，如图1-18所示。

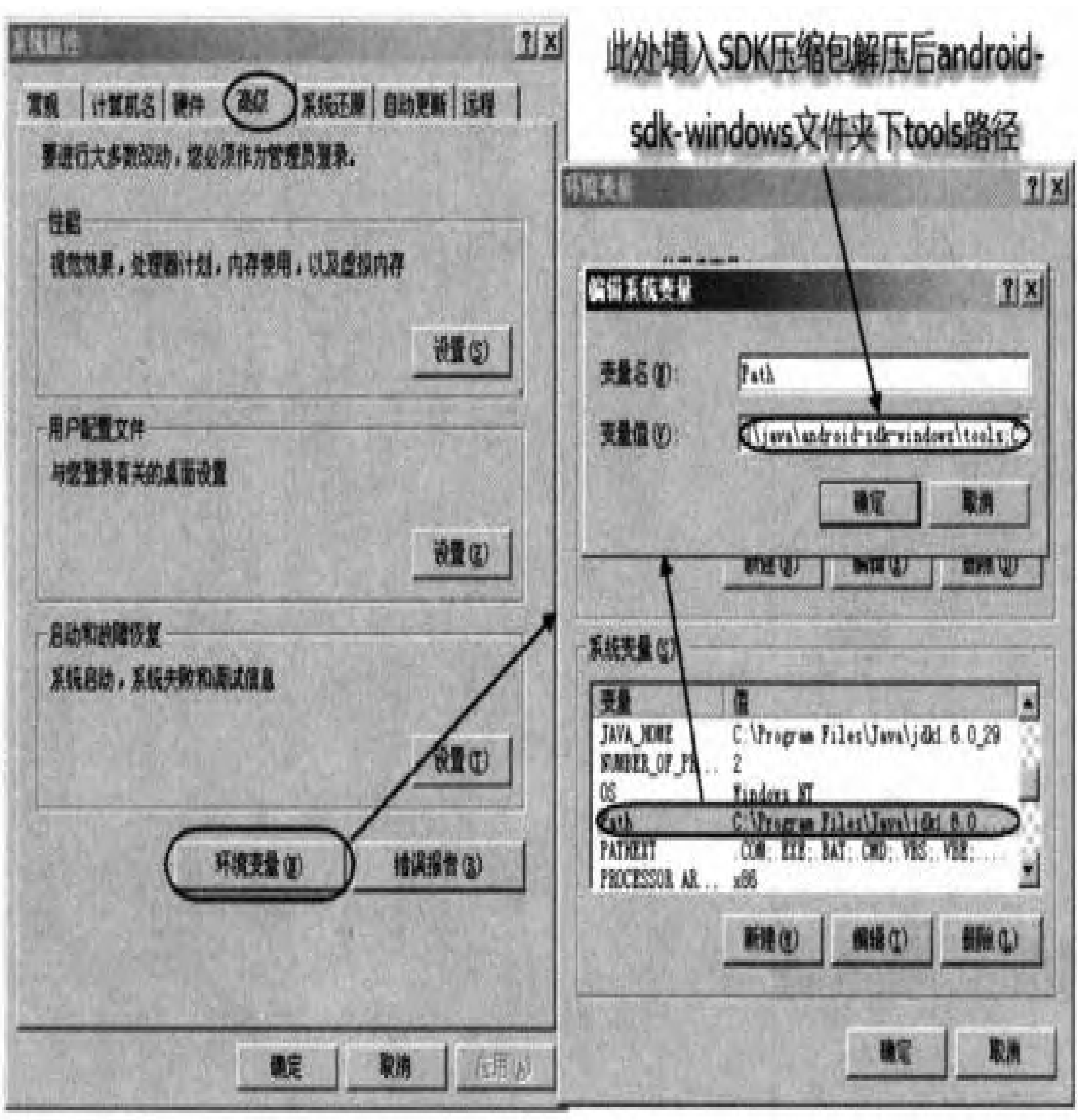


图1-18　设置SDK环境变量

提示

上述步骤中用到的路径都是基于笔者自己的计算机，若读者的计算机路径有所不同则需要在配置时作出相应的修改。同时，上述给出的是Windows下的安装配置过程，其他操作系统平台的安装与之类似，读者可以依此类推或查阅相关资料辅助完成。

1.3.2 Eclipse集成开发环境的搭建

完成了相关SDK的安装及环境变量的配置后，接下来就可以安装并配置Eclipse集成开发环境了，具体步骤如下。

- 首先登录Eclipse官方网站http://www.eclipse.org/downloads/，下载Eclipse的安装zip压缩包，如eclipse-java-indigo-SR1-win32.zip。
- 将下载完的Eclipse zip压缩包解压到硬盘的某个位置，打开解压后的文件夹，双击其中的eclipse.exe可执行文件，启动Eclipse。
- 在Help菜单中选择Install New Software，进入Install界面，如图1-19所示。

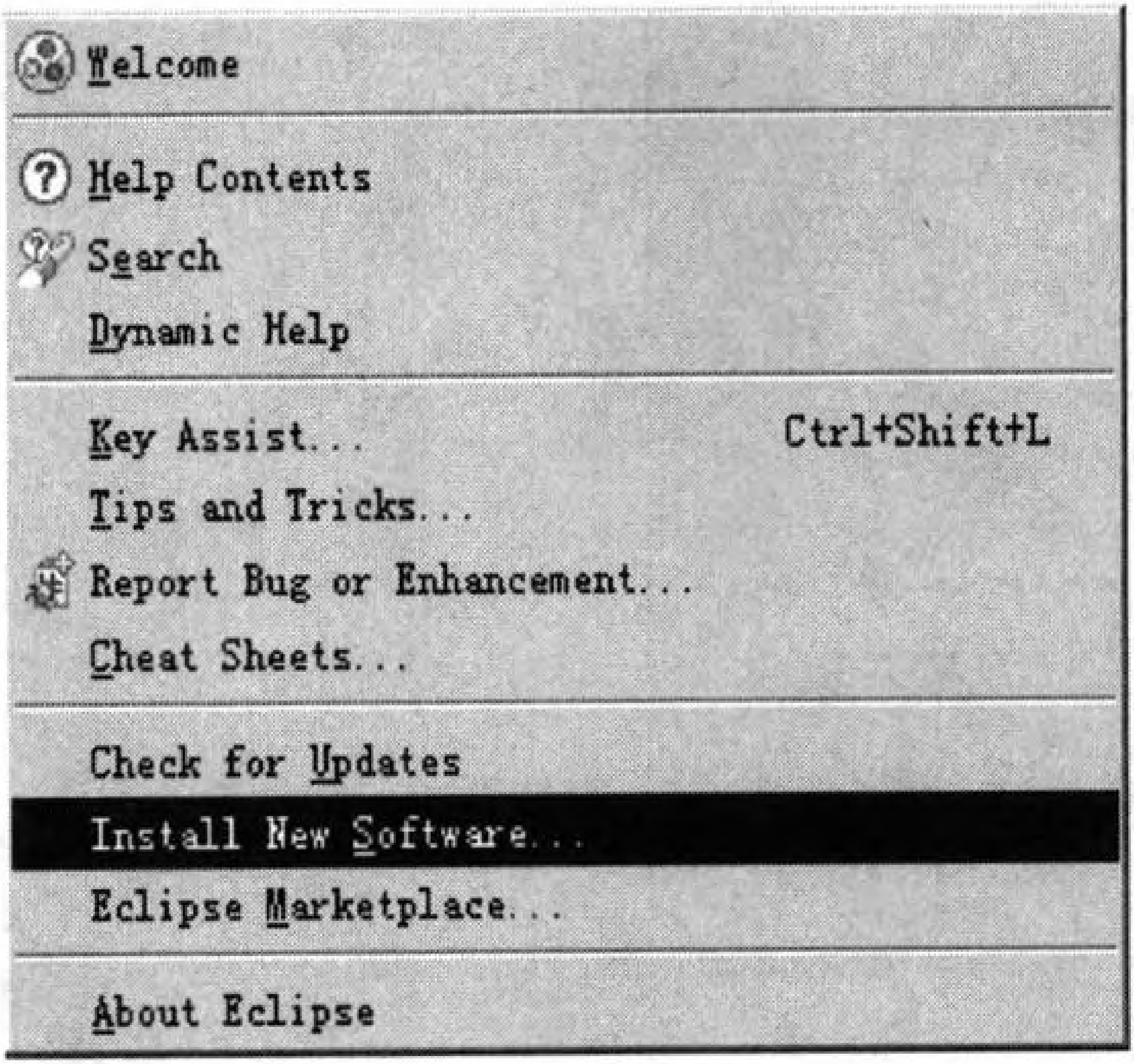


图1-19　选择Install New Software选项

- 接下来安装ADT（Android Development Tools）插件，在Install界面中单击Add按钮，在Name输入框中可输入任意名称，这里输入的是android，在Location输入框中输入ADT插件的安装URL:http://dl-ssl.google.com/android/eclipse/，输入完成后单击“OK”按钮，如图1-20所示。

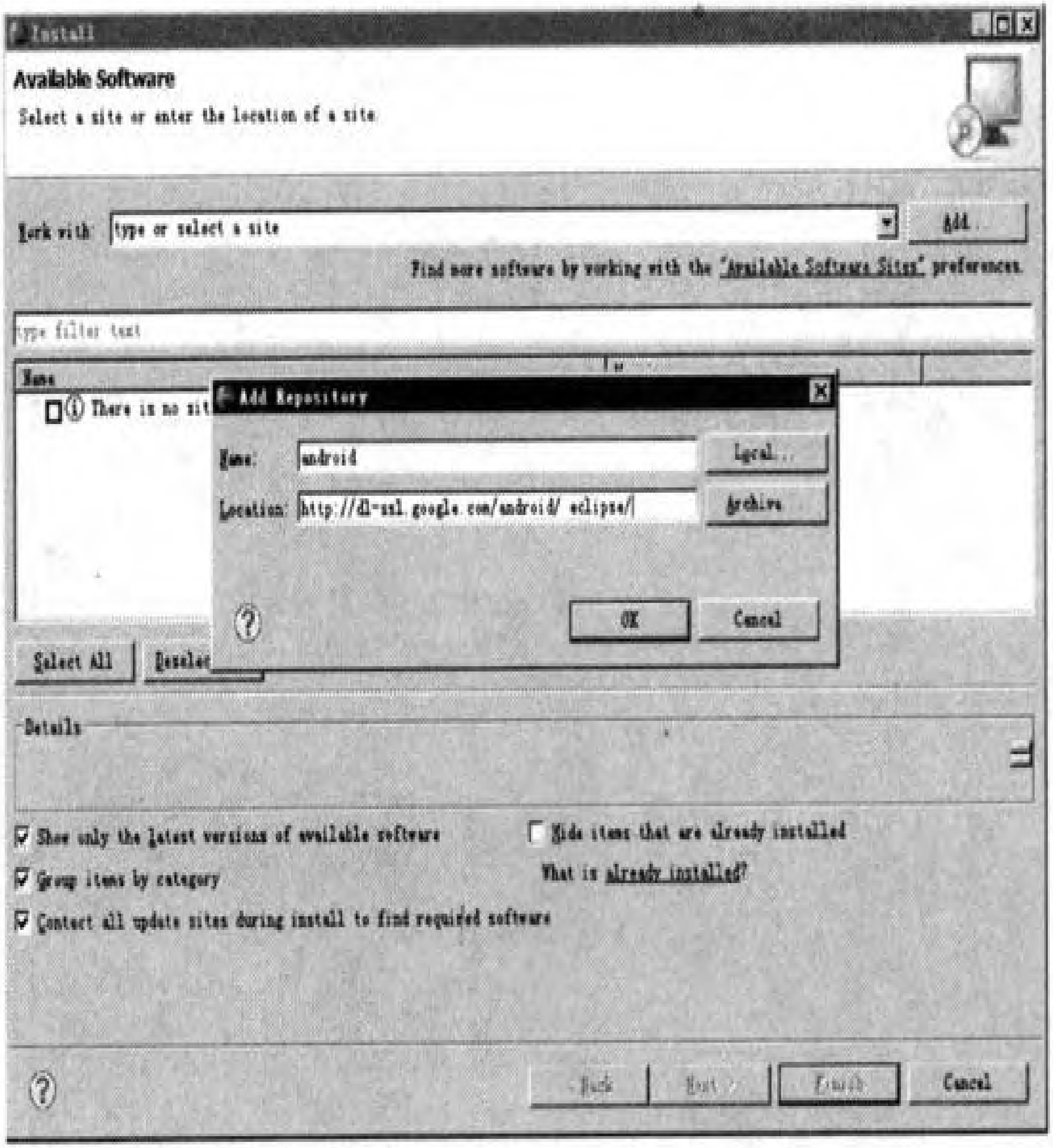


图1-20　Install界面

• 等待一会儿以后，Install界面中会列出可以安装的项目。在其中选中Developer Tools之后单击“Next”按钮，系统将自动联网下载安装，此时要保证计算机是可以接入互联网的。

• 等待安装完成后，在弹出的界面中，直接单击“Finish”按钮，重启Eclipse，即完成了ADT插件的安装。

- Eclipse重新启动后，选择菜单中的Window选项并选中其下的Preferences选项，打开Preferences界面。
- 在Preferences界面中首先选中左侧列表栏中的Android选项，然后单击右侧的“Browse” 按钮，添加Android SDK的安装路径（此安装路径需要与上一小节中的匹配），如图1-21所示。

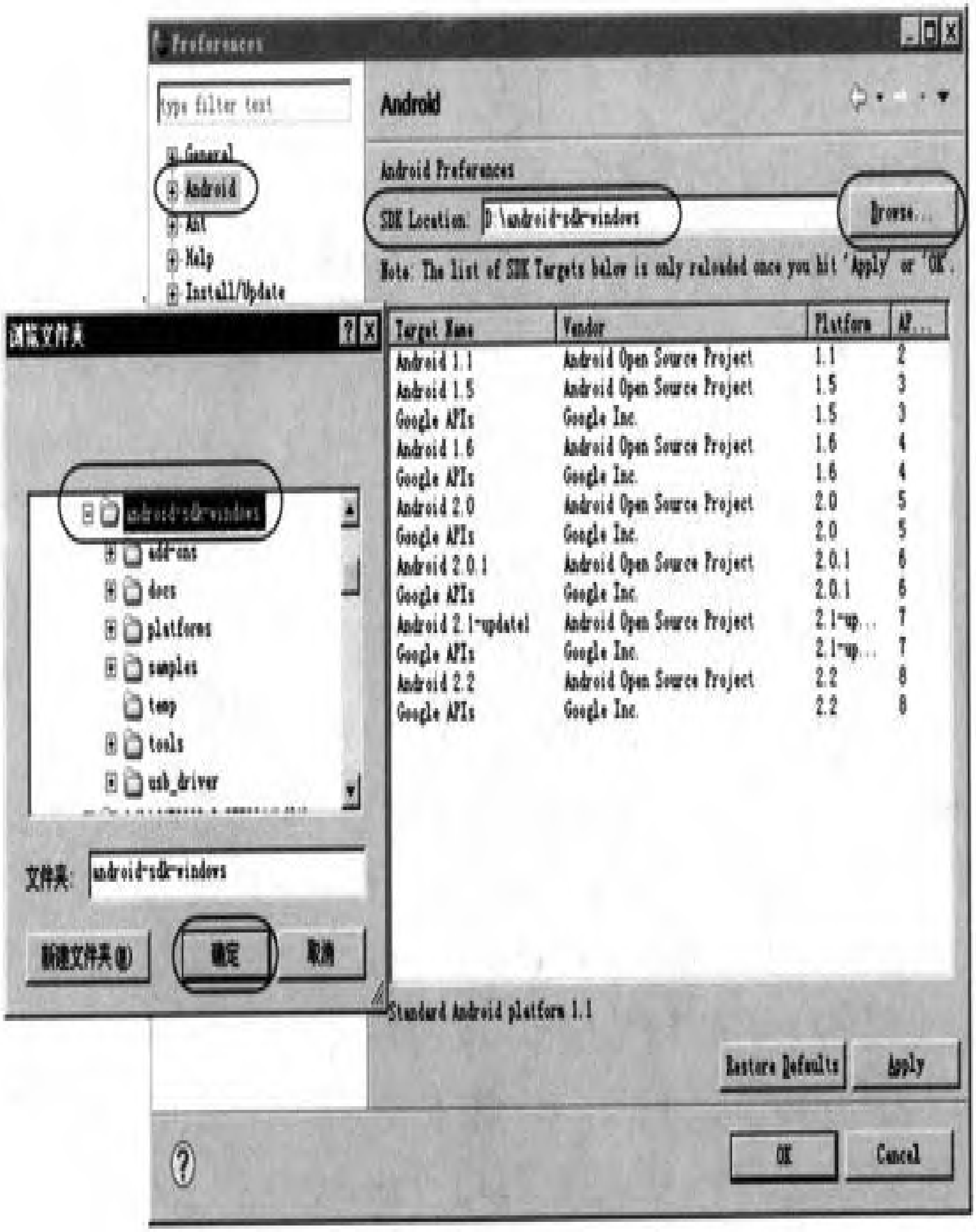


图1-21　Preferences界面

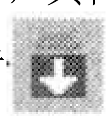
- 添加完了Android SDK的安装路径后，单击“OK”按钮，结束Eclipse的环境配置，此时就可以正式开始Android应用开发之旅了。

> **提示**
>
> 上述给出的是Windows下的安装配置过程，其他操作系统平台的安装与之类似，读者可以依此类推或查阅相关资料辅助完成。

1.3.3 创建并启动模拟器

开发环境搭建基本完成后。在正式开发Android应用程序之前，还有一个很重要的工作就是创建模拟器，模拟器可以在初学者没有实际设备的情况下在PC上对应用程序进行运行测试，很大程度上降低了学习的成本。

创建模拟器的步骤很简单，具体如下所列。

- 首先在Eclipse中单击（Opens the Android SDK and AVD Manager）按钮，如图1-22所示。

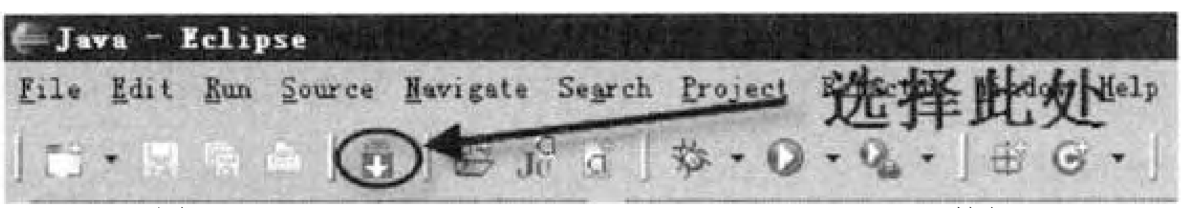
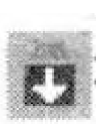


图1-22 Opens the Android SDK and AVD Manager按钮

- 按下按钮后，系统将弹出Android SDK and AVD Manager对话框，如图1-23所示。

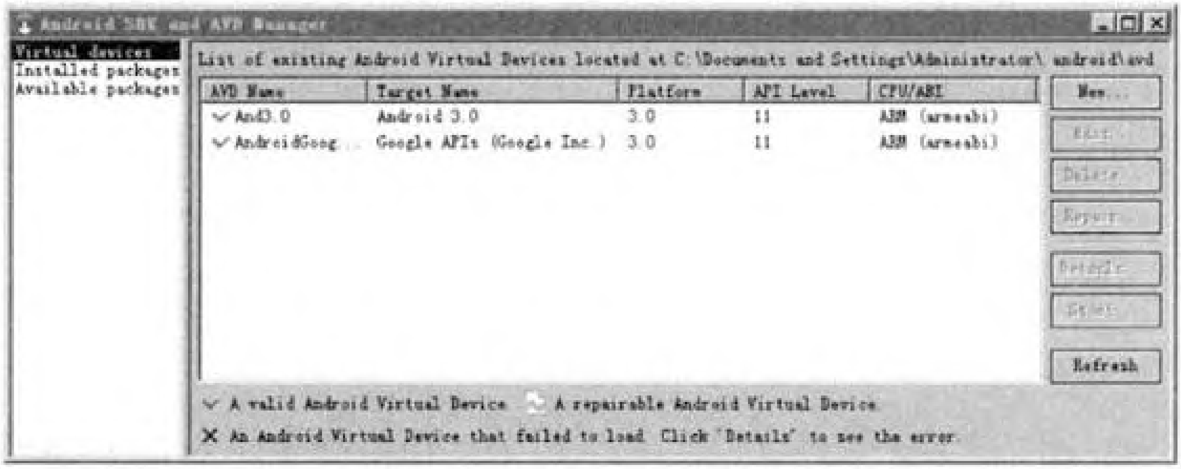


图1-23　Android SDK and AVD Manager对话框

• 在弹出的Android SDK and AVD Manager对话框中单击“New”按钮（如图1-24所示），系统将弹出Create new Android Virtual Device（AVD）对话框，如图1-25所示。

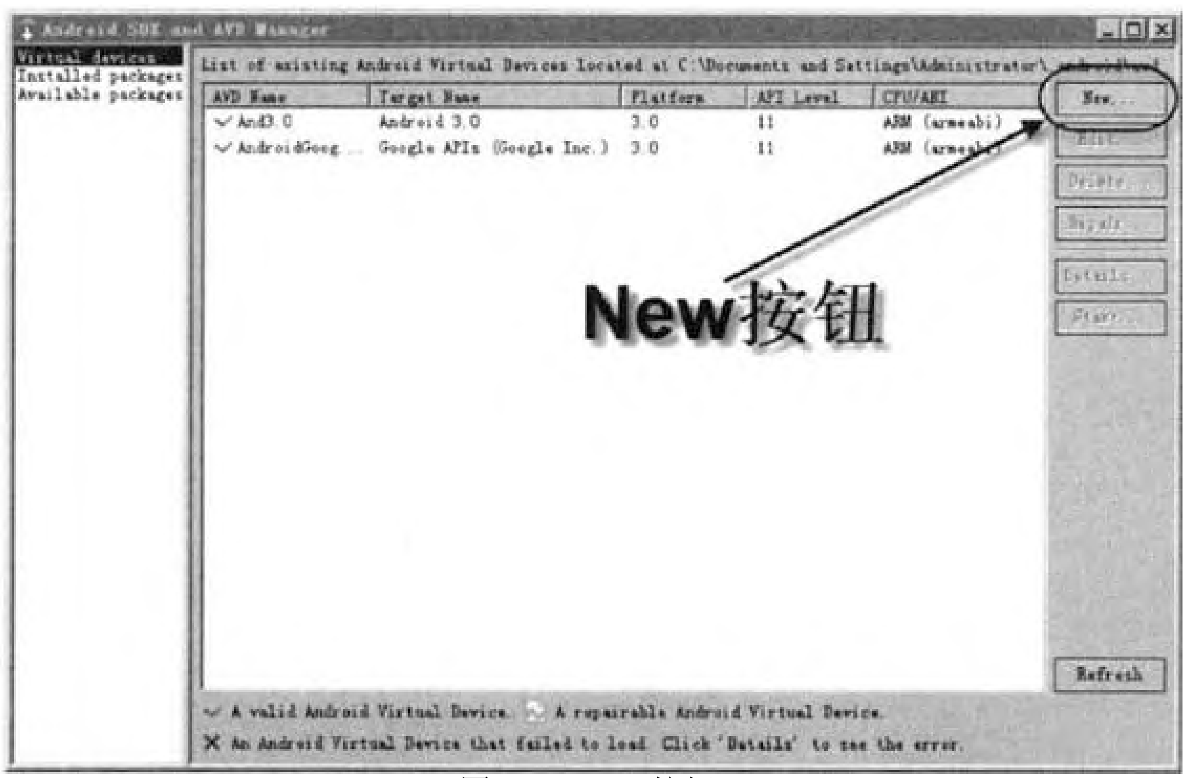


图1-24　New按钮

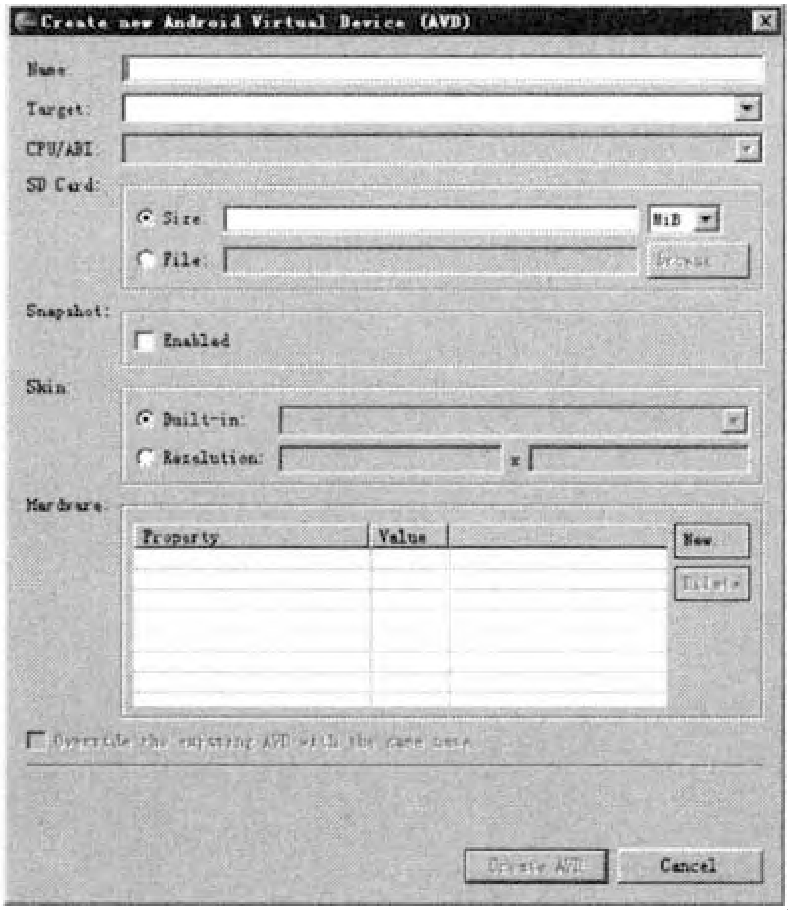


图1-25　Create new Android Virtual Device（AVD）对话框

- 在Create new Android Virtual Device（AVD）对话框中输入模拟器的名称（这里输入的是And2.3），下拉Target列表选中Android 2.3.1-API Level 9，在SD Card面板中的Size输入框中输入100（代表SD卡容量为100MB），同时设置模拟器显示屏分辨率为HVGA（480×320），如图1-26所示。

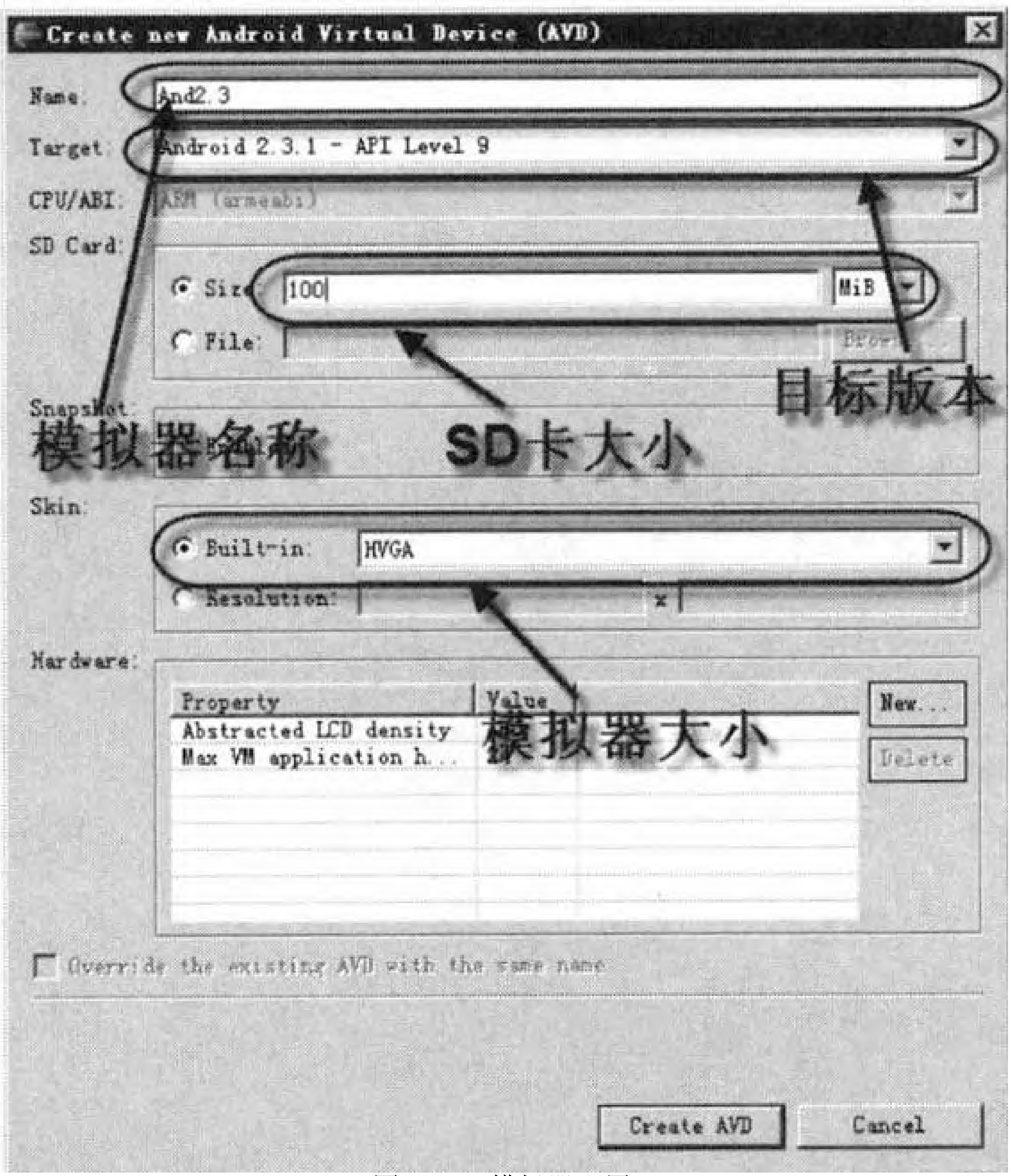


图1-26　模拟器配置

• 模拟器配置完成后，单击“Create AVD”按钮（如图1-27所示），即可创建指定版本Android的模拟器。创建完Android的模拟器后，在Android SDK and AVD Manager对话框中就可以显示出创建的Android模拟器了，如图1-28所示。

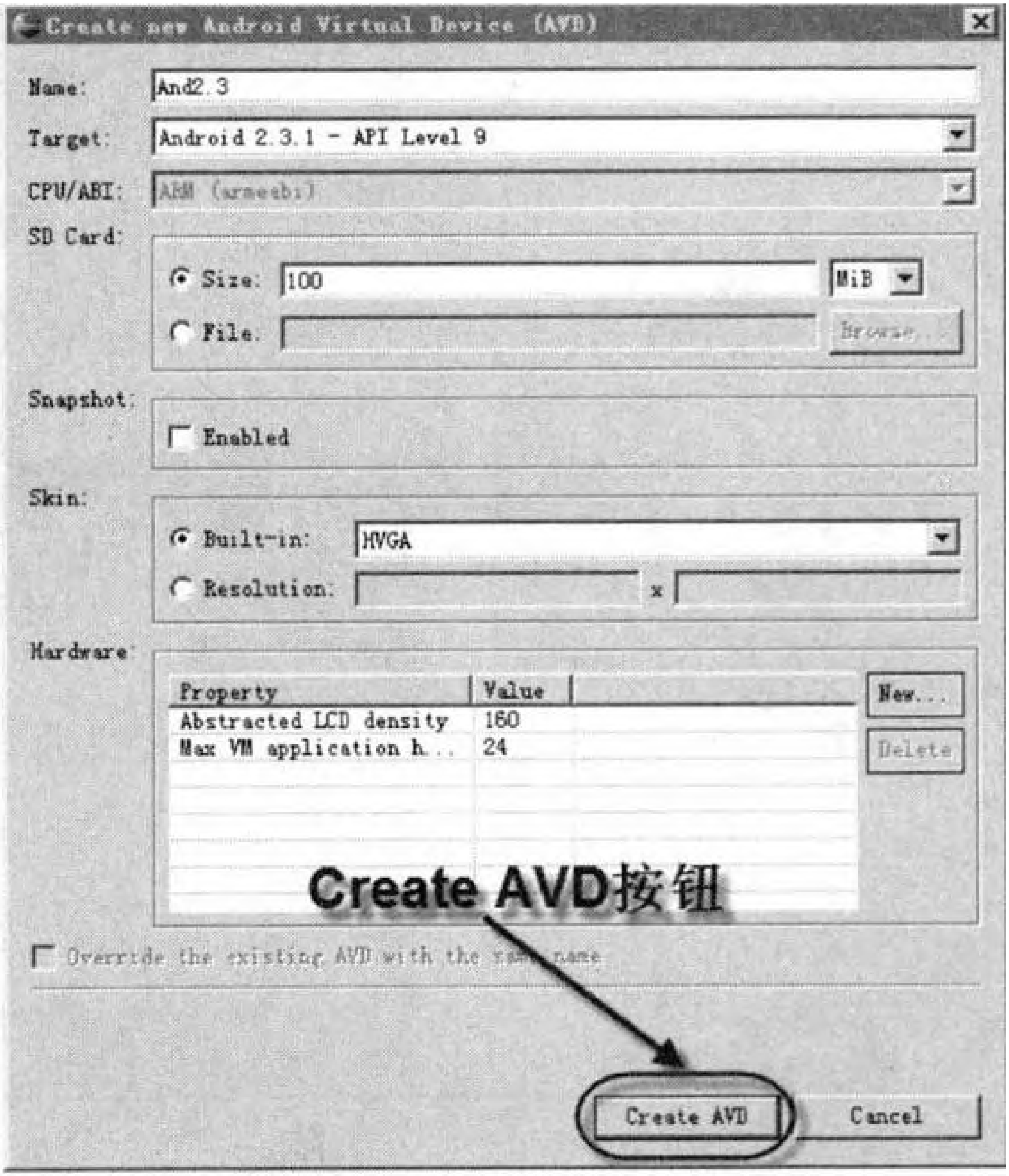


图1-27　Create AVD按钮



图1-28　创建完成的Android SDK模拟器

• 在Android SDK and AVD Manager对话框中选中创建的Android模拟器，单击“Start”按钮 （如图1-29所示），弹出Launch Options对话框，如图1-30所示。

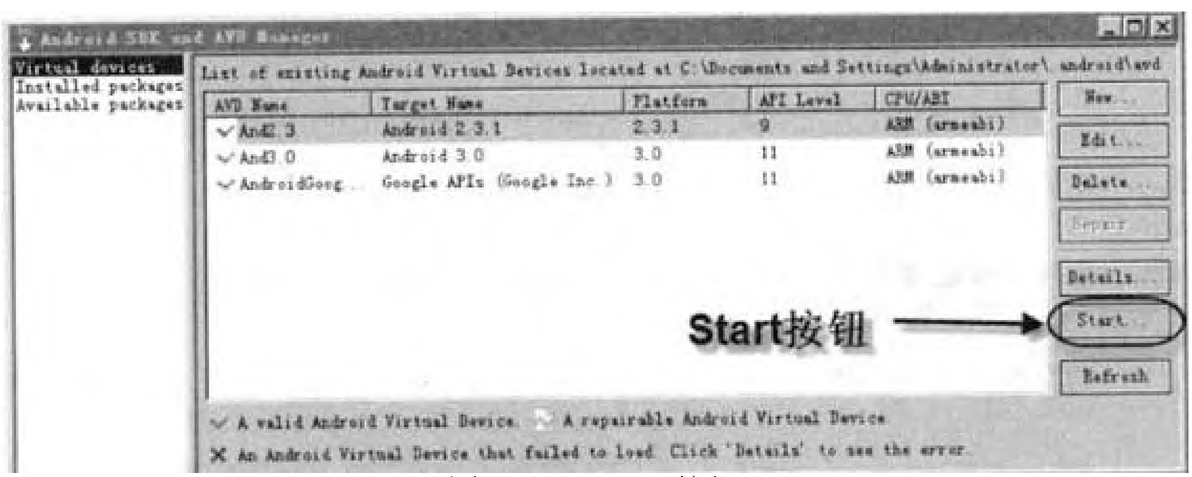


图1-29　Start按钮

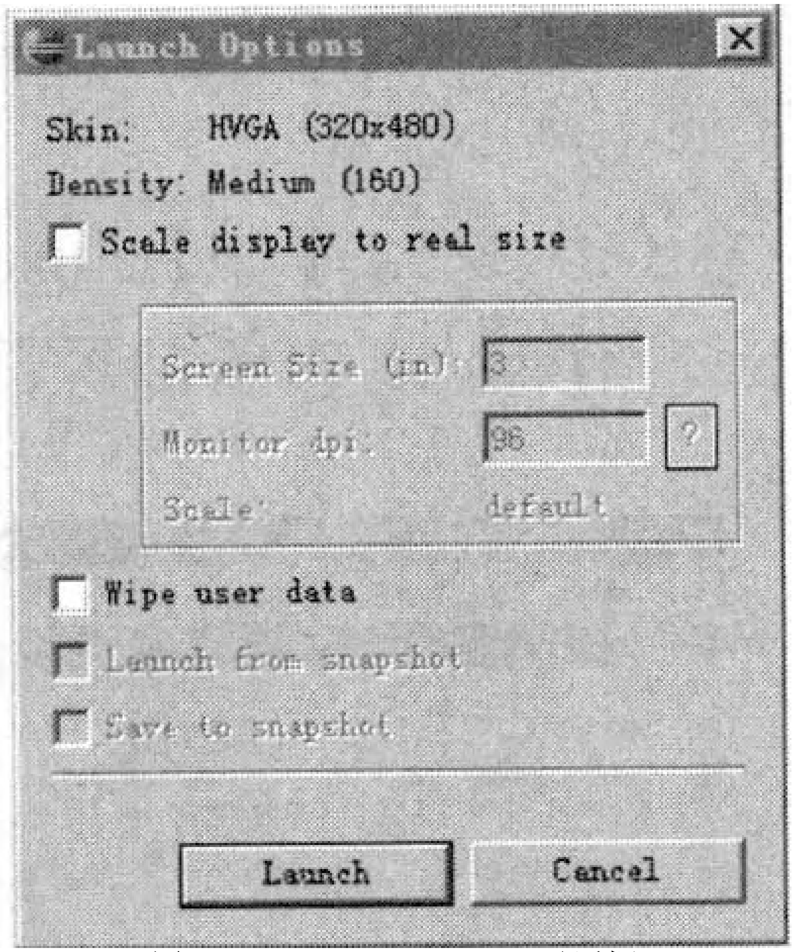


图1-30　Launch Options对话框

- 在弹出的Launch Options对话框中单击“Launch”按钮（如图1-31所示），系统将启动Android模拟器，启动完成后的效果如图1-32所示。



图1-31　Launch按钮

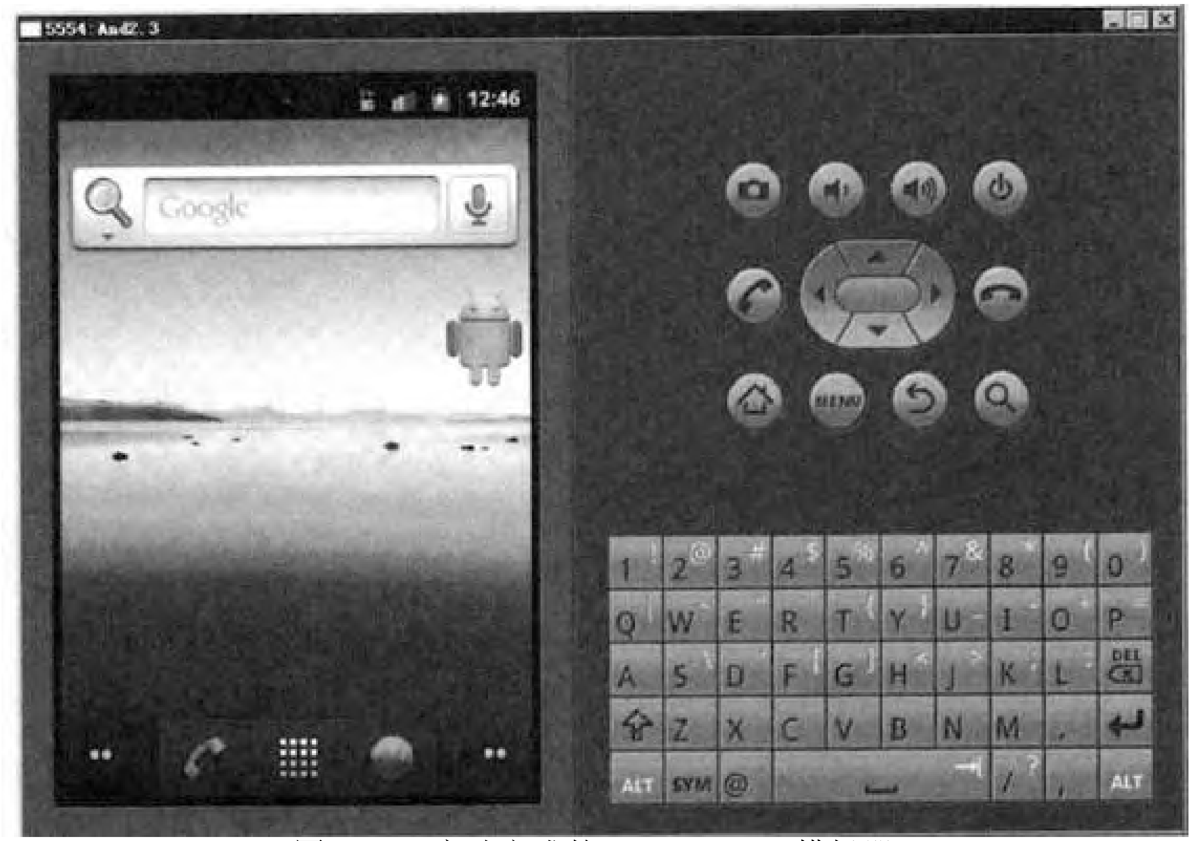


图1-32　启动完成的Android SDK模拟器

> **提示**
>
> 由于模拟器性能不是很好，启动可能需要几分钟的时间，这时请不要着急，喝杯咖啡吧。

1.4　Hello Android应用程序的开发

本节首先将介绍如何在Eclipse中创建一个基于Android的Hello World应用程序，之后将简单介绍Android应用程序的调试，为读者以后学习高级开发铺平道路。

1.4.1　第一个Android应用程序

本小节将向读者介绍如何在Eclipse中创建一个基于Android的Hello World应用程序，基本步骤如下所列。

（1）首先打开Eclipse，然后依次选择File|New|Project（如图1-33所示）或者单击按钮后，系统将弹出新建项目对话框，如图1-34所示。

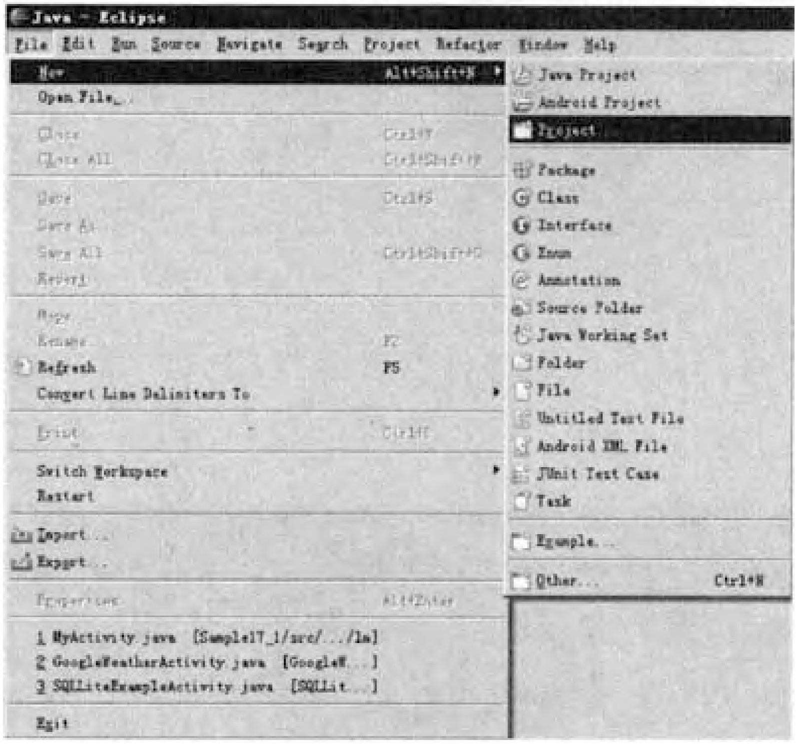


图1-33　新建项目方法

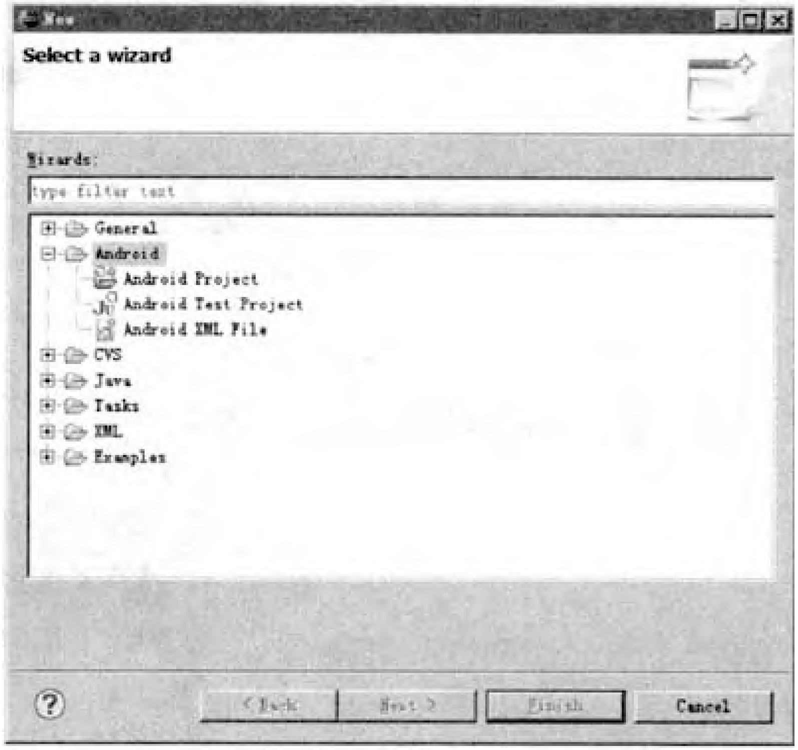


图1-34　新建项目对话框

（2）在新建项目对话框中选择Android Project，然后单击“Next”按钮（如图1-35所示），弹出New Android Project对话框，如图1-36所示。



图1-35　选择Android Project创建Android应用程序

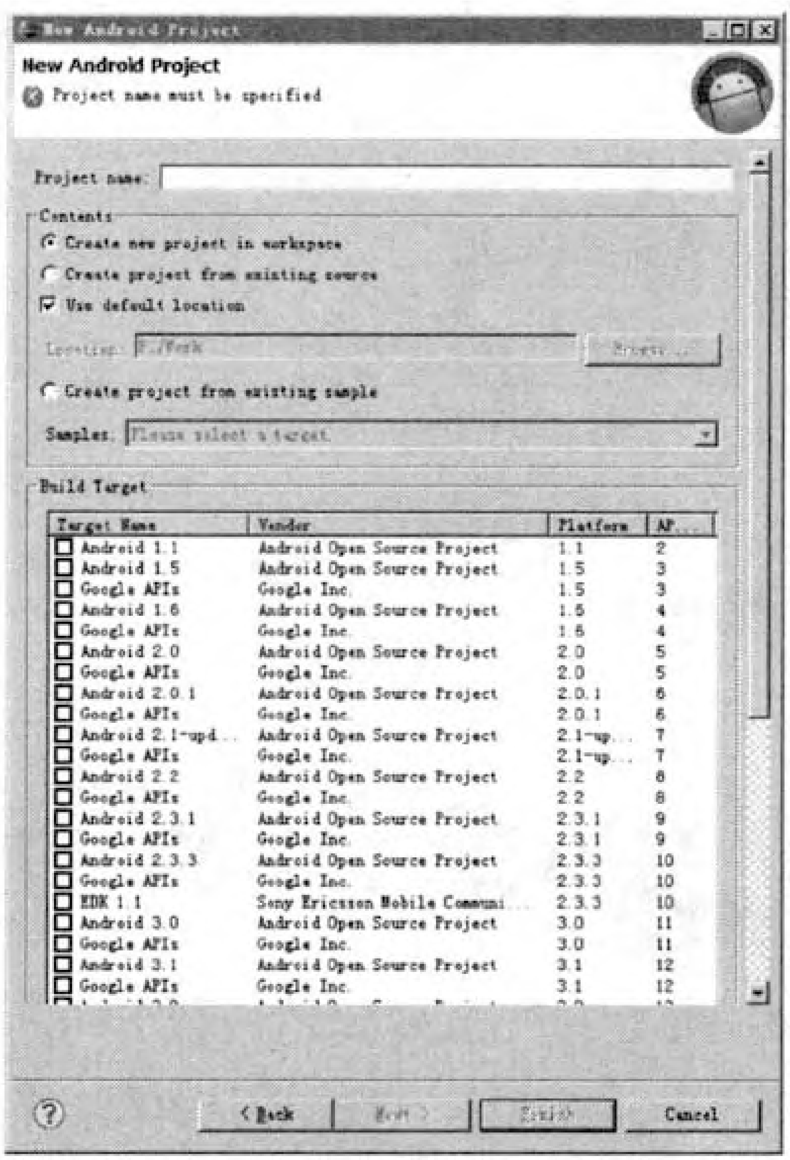


图1-36　New Android Project对话框

（3）在New Android Project对话框中输入项目的名称（在这里输入的是Samplel_1），在Build Target中选择项目的目标版本（这里选择的是Android 2.3.1版），如图1-37所示。

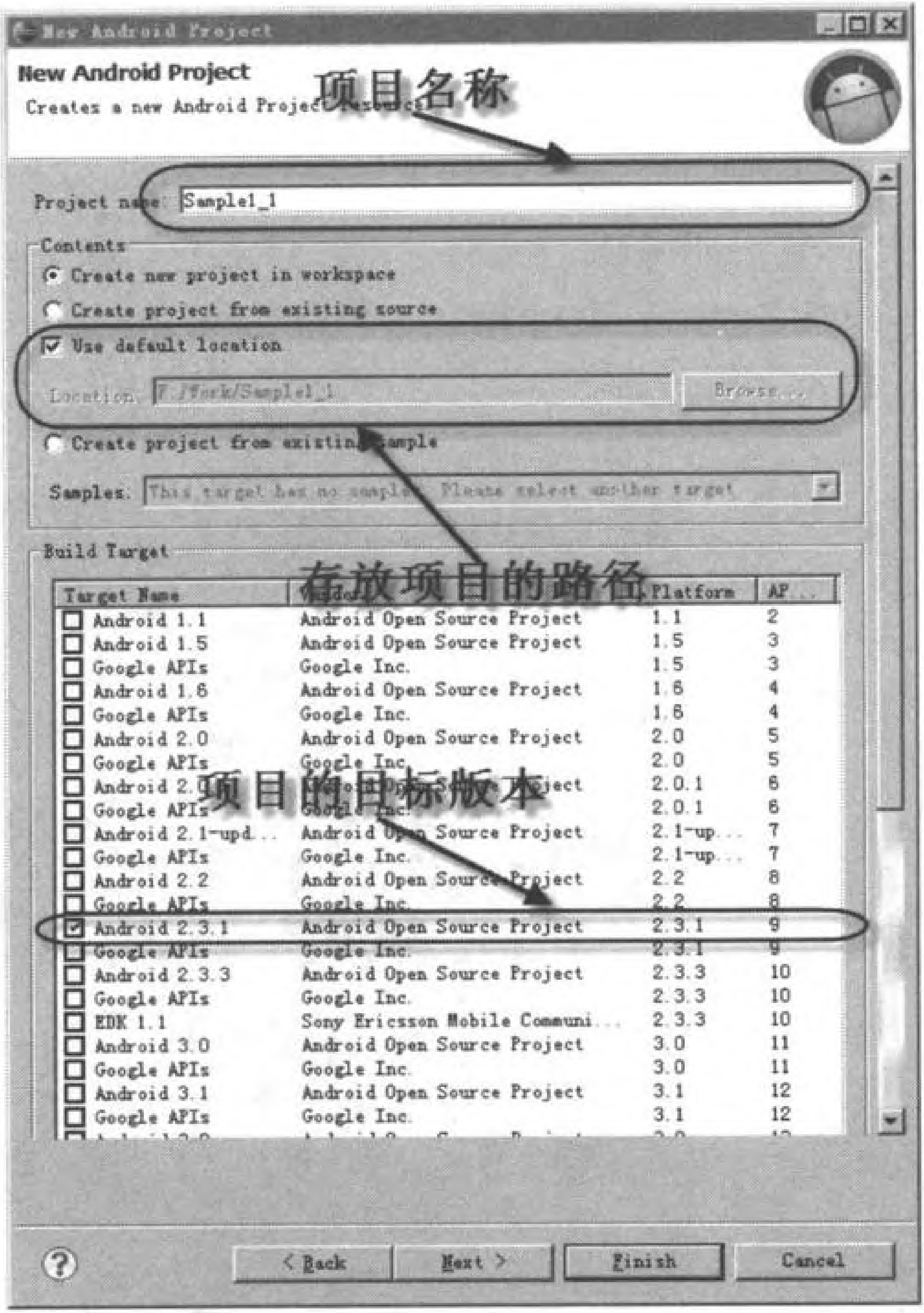
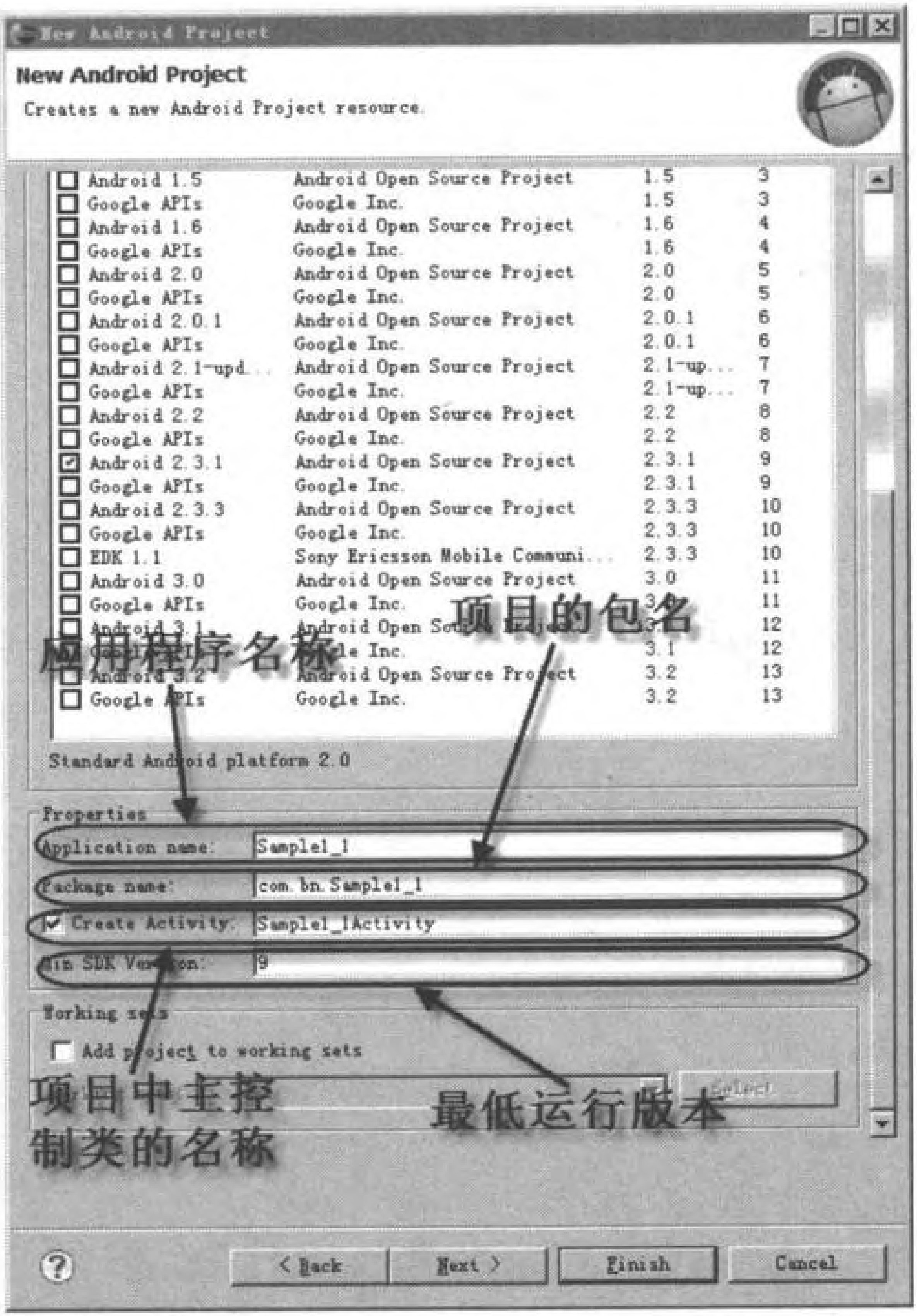

New Android Project
Creates a new Android Project resource.
项目名称
Project name: Sample1_1
Contents
Create new project in workspace
Create project from existing source
Use default location
Location: F:/Work/Sample1_1
Browse...
Create project from existing sample
Samples: This target has no samples. Please select another target.
Build Target
存放项目的路径
项目的目标版本
Target Name
Platform
Android 1.1
Android Open Source Project
Android 1.5
Google APIs
Google Inc.
Android 1.6
Android 2.0
Android 2.1-upd...
Android 2.2
Android 2.3.1
Android 2.3.3
EDK 1.1
Sony Ericsson Mobile Communi...
Android 3.0
Android 3.1
< Back
Next >
Finish
Cancel

图1-37　创建项目截图1

（4）接着在Application name中输入应用程序的名称（这里输入的名称与Project Name中的相同，当然也可以不同）；同时在Package name中输入程序的包名（这里输入的是com.bn.Sample1_1）；最后输入Activity的名称（在这里输入的是Sample1_1_Activity）；最后在Min SDK Version中输入应用程序最低支持的版本Level（这里输入的是9，代表Android 2.3.1版）。整体情况如图1-38所示。



New Android Project
Creates a new Android Project resource.
Android 1.5	Android Open Source Project	1.5	3
Google APIs	Google Inc.	1.5	3
Android 1.6	Android Open Source Project	1.6	4
Google APIs	Google Inc.	1.6	4
Android 2.0	Android Open Source Project	2.0	5
Google APIs	Google Inc.	2.0	5
Android 2.0.1	Android Open Source Project	2.0.1	6
Google APIs	Google Inc.	2.0.1	6
Android 2.1-upd...	Android Open Source Project	2.1-up...	7
Google APIs	Google Inc.	2.1-up...	7
Android 2.2	Android Open Source Project	2.2	8
Google APIs	Google Inc.	2.2	8
Android 2.3.1	Android Open Source Project	2.3.1	9
Google APIs	Google Inc.	2.3.1	9
Android 2.3.3	Android Open Source Project	2.3.3	10
Google APIs	Google Inc.	2.3.3	10
EDK 1.1	Sony Ericsson Mobile Communi...	2.3.3	10
Android 3.0	Android Open Source Project	3.0	11
Google APIs	Google Inc.	11
12
3.1	12
Android 3.2	Android Open Source Project	3.2	13
Google APIs	Google Inc.	3.2	13
项目的包名
应用程序名称
Standard Android platform 2.0
Properties
Application name:	Sample1_1
Package name:	com.bn.Sample1_1
Create Activity:	Sample1_1Activity
Min SDK Version:	9
Working sets
Add project to working sets
项目中主控
制类的名称
最低运行版本
< Back
Next >
Finish
Cancel

图1-38　创建项目截图2

提示

创建项目时输入项目的包名时，需要至少使用二级包名，否则在该对话框上方会显示“Package name must have at least two identifiers”错误信息。

（5）最后单击“Finish”按钮（如图1-39所示）完成项目的创建，此时在Project Explorer中会自动添加创建的项目，如图1-40所示。

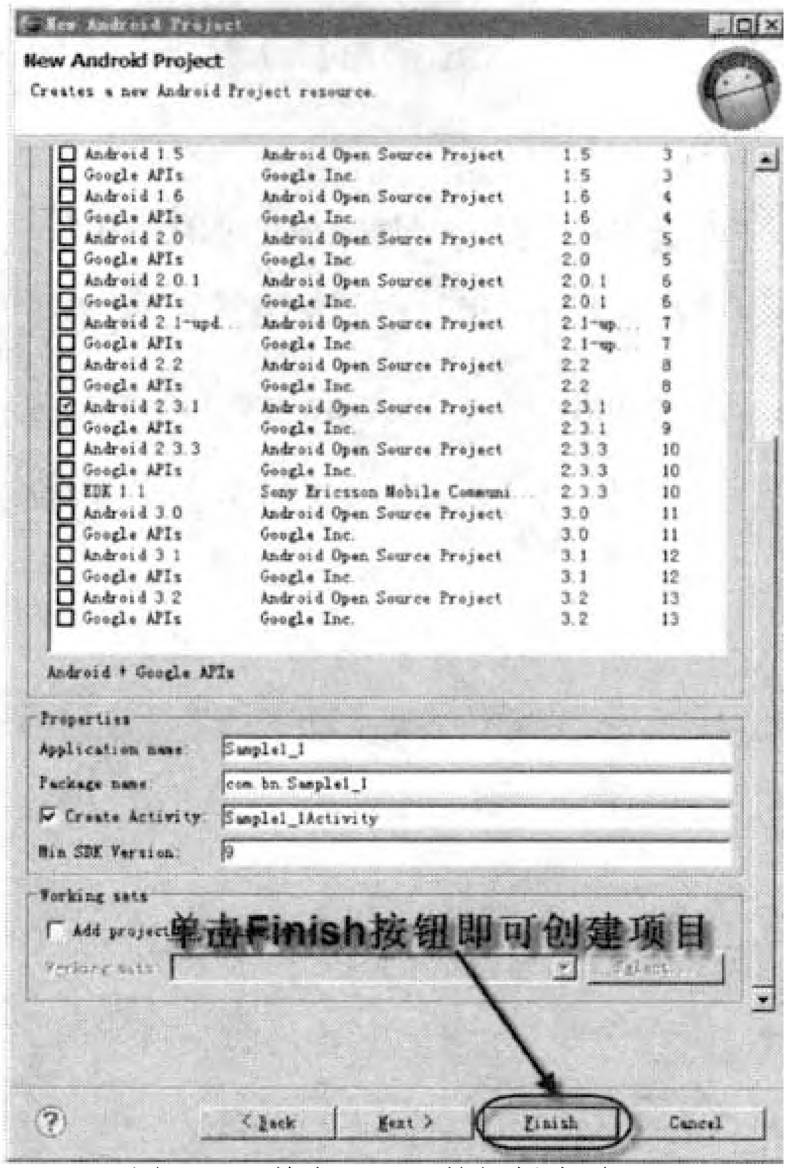


图1-39　单击Finish按钮创建项目

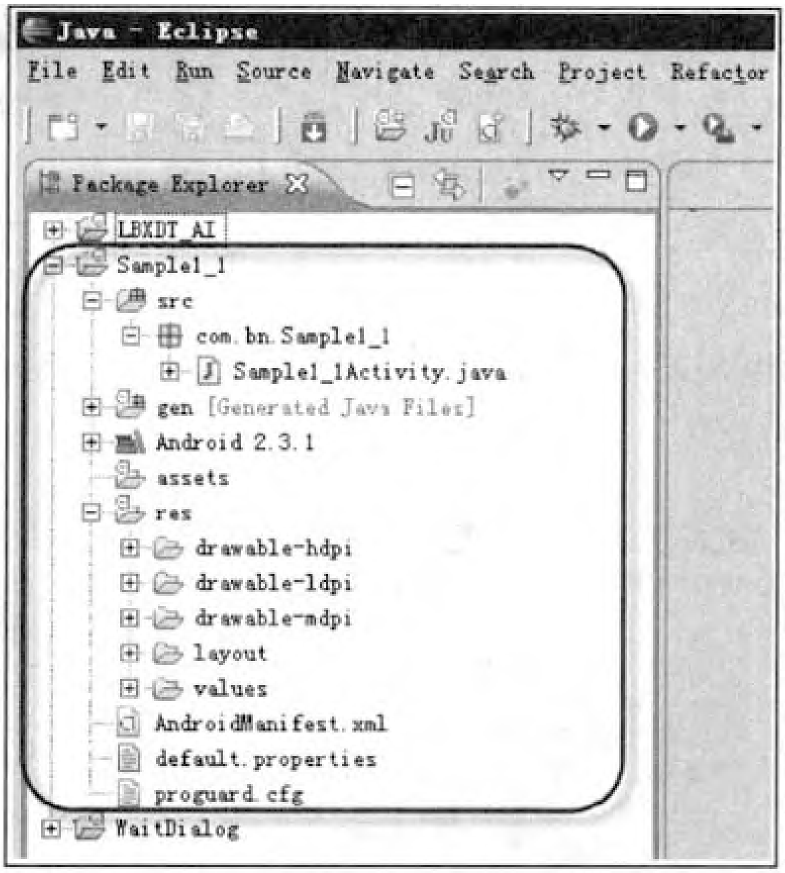
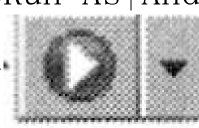


图1-40　创建完成的Android项目

（6）在Project Explorer中选择本项目，然后单击鼠标右键，在弹出的快捷菜单中依次选择Run As|Android Application运行本程序（如图1-41所示）或者单击按钮运行本程序，运行效果如图1-42所示。

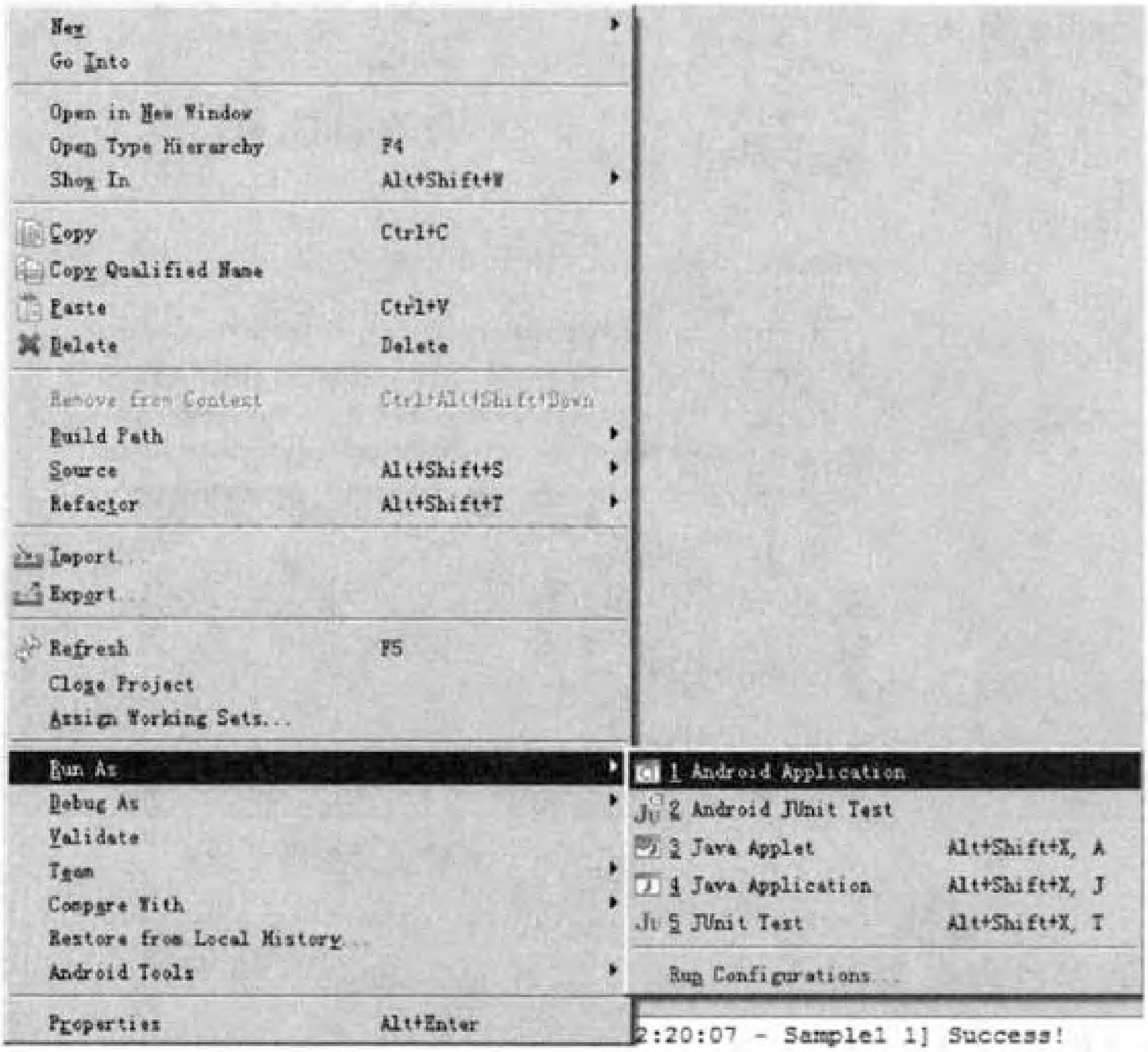


图1-41　运行Android程序方式1

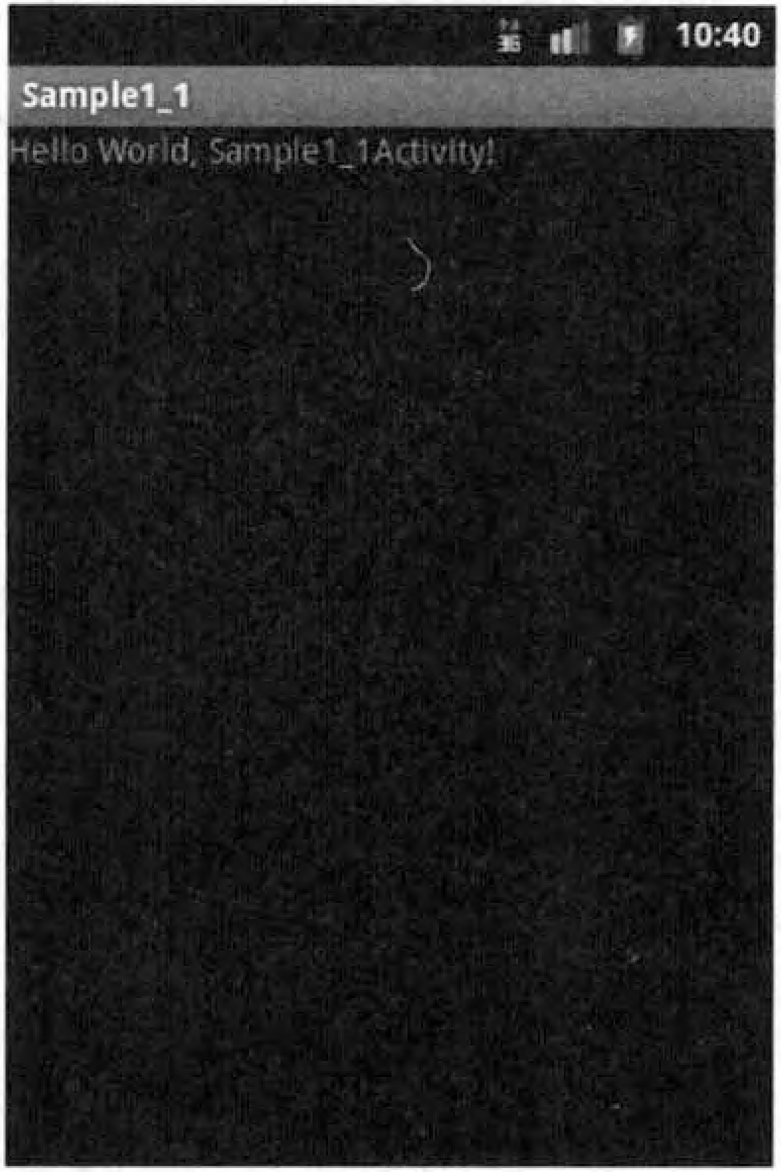


图1-42　运行效果图

从上述步骤中可以看出，使用Eclipse创建并运行一个Android应用程序是非常容易的。

1.4.2　Android应用程序的项目结构

上一小节介绍了如何通过Android SDK自动生成Android项目，但是没有对该项目中的内容及组成进行介绍。本小节将对Android项目中的各重要组成部分逐一进行介绍，如图1-43所示。



图1-43　Android应用程序项目结构

1．src源代码文件夹

该文件夹下存放了应用程序中所有开发的Java源代码。要注意的是，在更为复杂的项目中，该文件夹下可以有很多不同的包用于存放不同功能目的的源代码。

2．gen文件夹

该文件夹中存放的是Eclipse中ADT插件自动生成的R.java文件。实际上R.java定义了一个R类，其中包含应用程序的用户界面、图像以及字符串等资源的编号。也就是每有一个资源实例，系统就为其在R类中生成一个相应的编号。这些编号都是系统自动生成的，用户不应该修改。

3．assets文件夹

该文件夹用于存放应用程序需要使用的一些外部资源，如音频、视频、数据文件等。在应用程序中可以使用

“getResources（）.getAssets（）.open（″aa.mp3″）”得到指定资源文件的输入流。

4. res资源文件夹

该文件夹下一般有drawable系列、layout及values等文件夹，下面对这些文件夹逐一进行介绍。

- drawable系列文件夹。该系列文件夹一般有drawable-hdpi、drawable-mdpi、drawable-ldpi 3个文件夹，这3个文件夹分别用来存放不同分辨率目标设备的图片资源。当应用程序在目标设备上运行时，系统会自动根据目标设备的分辨率选择对应文件夹下的图片使用。
- layout文件夹。该文件夹下包含了所有使用xml格式的界面描述文件，这些文件用于表述应用程序中用户界面的布局。具体的开发中读者会体会到，用这种方式进行布局的开发比传统的使用Java源代码开发布局要方便简捷得多。
- values文件夹。本文件夹中包含了一些xml格式的资源描述文件，一般包括string.xml（字符串资源）、color.xml（颜色资源）、style.xml（样式资源）等。

5. AndroidManifest.xml文件

此文件是整个应用程序的描述文件，应用程序中所需要的各个组件均需要在此文件中说明。同 时，应用程序需要的一些权限，如网络访问权限、I/O读取权限等，也需要在此文件中进行声明。

> **提示**
>
> 除了上述文件夹及文件外，还有一个一般开发时不需要直接使用的default.properties文件。此文件为项目的配置文件，一般不需要人为改动，系统会自动对其进行管理，有兴趣的读者可以自行打开查看。

1.4.3 Android应用程序的调试

Android SDK提供了一个强大的调试工具DDMS（Dalvik Debug Monitor Service），通过DDMS可以调试并监控Android程序的运行，打开DDMS的方法如下。

- 一般情况下在Eclipse中安装了ADT插件后，Eclipse窗口的右上侧会有DDMS的选项按钮  ，如图1-44所示。单击它即可切换到DDMS界面，如图1-45所示。

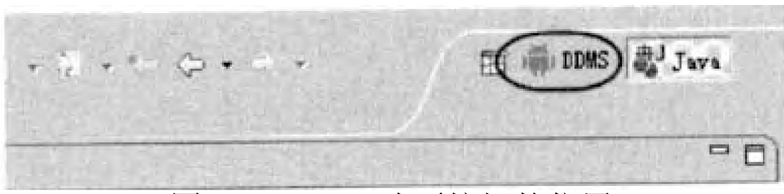


图1-44　DDMS选项按钮的位置

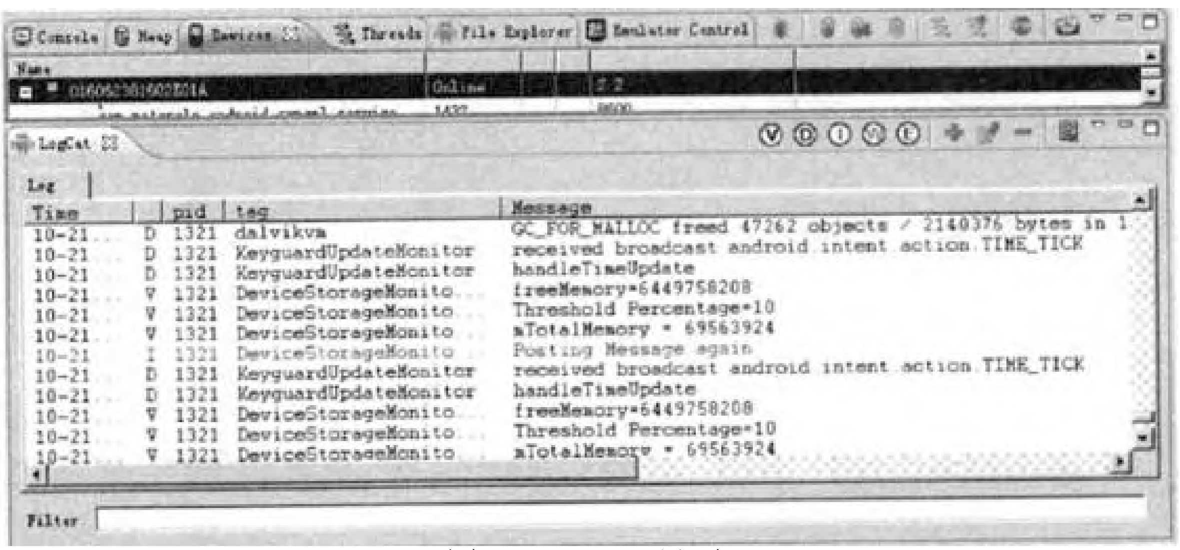
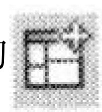


图1-45　DDMS界面

- 有些时候安装完ADT插件后，DDMS选项按钮  是隐藏的，读者可以通过Eclipse窗口界面右上侧的  （Open Perstective）按钮来打开DDMS，如图1-46所示。

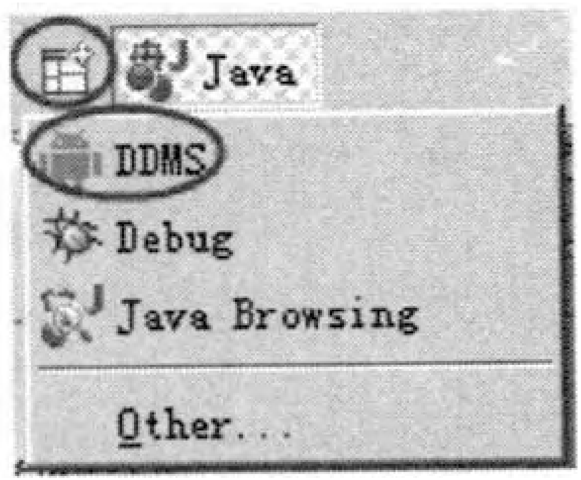


图1-46　使用Open Perstective按钮打开DDMS

从前面的介绍中可以看出，DDMS的一大功能就是查看程序运行时的后台输出信息。实际的应用程序开发中既可以使用传统的System.out.println方法来打印输出调试信息，也可以使用Android特有的android.util.Log类来输出调试信息，这两种方法的具体使用方式如下。

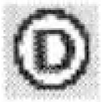
1．System.out.println方法

首先介绍Java开发人员十分熟悉的System.out.println方法，其在Android应用程序中的使用方法与传统Java中相同，具体步骤如下。

> **提示**
>
> 在这里就不再创建新的Android项目了，直接使用的是上一小节已经创建的Android项目（Sample1_1）。

- 首先在Eclipse中打开src文件夹下的com.bn.Sample1_1包中的Sample1_1Activity.java文件。
- 然后在setContentView（R.layout.main）语句后面添加代码System.out.println（“the first”）。
- 代码修改完成后，再次运行本应用程序。

• 应用程序运行后打开DDMS，找到LogCat面板，单击其中的



（debug）按钮，如图1-47所示。



图1-47　DEBUG界面

• 在LogCat面板下的Log选项卡中可以看到输出的打印语句，如图1-48所示。

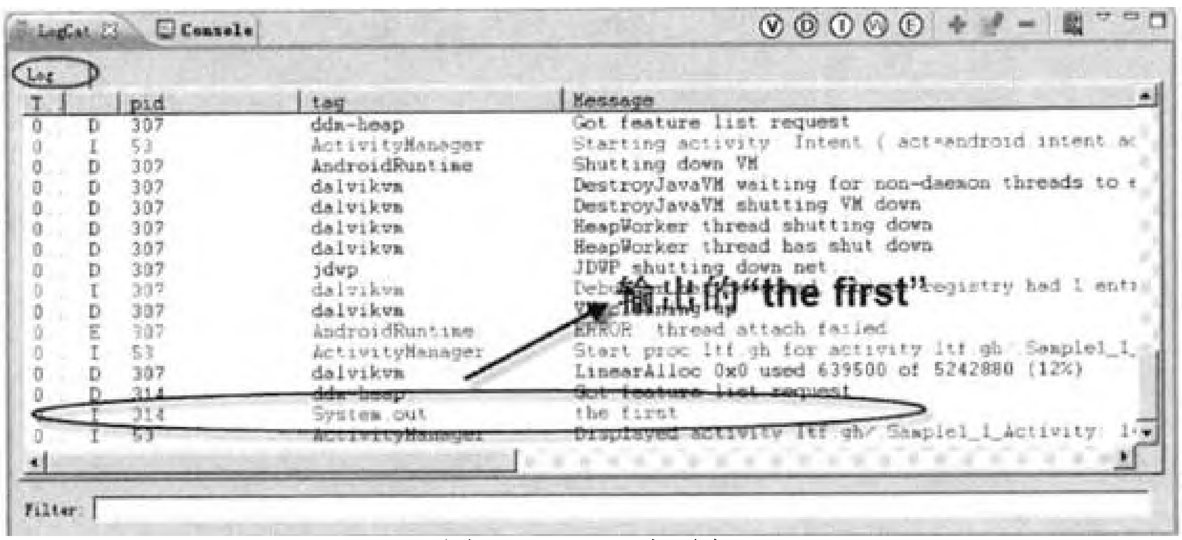
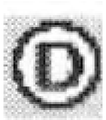
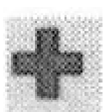


图1-48　Log选项卡

有时可能觉得在Log中的输出信息太多，不便于查看。这时可以在LogCat中添加一个专门输出System.out信息的面板。单击DEBUG按钮



右边的



（Create Filter）按钮，系统会弹出Log Filter对话框，在Filter Name输入框中输入过滤器名称，在by Log Tag中输入用于过滤的标志，如图1-49所示。

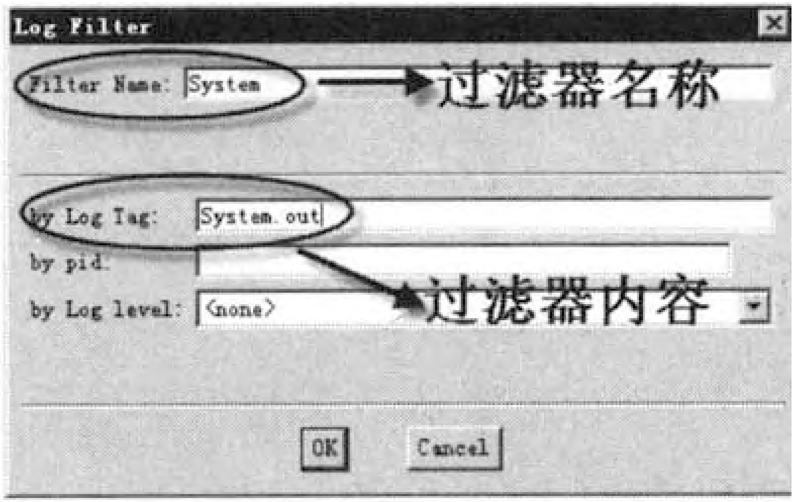
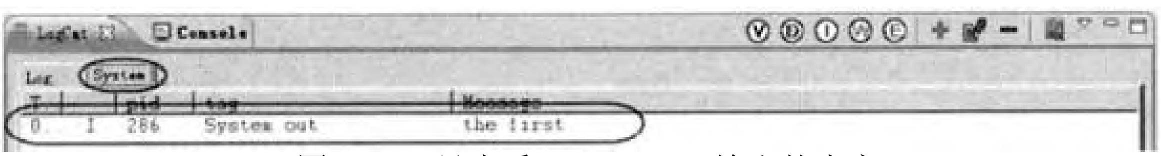


图1-49　Log Filter对话框

> **说明**
>
> 由于输出的语句主要有System.out.println（换行），System.out.print（不换行）两种，所以设置by Log Tag中的内容为System.out以进行过滤。

此时再次运行应用程序观察输出的情况，在LogCat下的System面板中将会只存在System.out的输出信息，效果如图1-50所示。


图1-50　只查看System.out输出的内容

2. android.util.Log类

除了Java开发人员熟知的System.out.println方法外，Android还专门提供了另外一个类android.util.Log来进行调试信息的输出。下面将介绍Log类的使用，具体步骤如下。

- 在Sample1_1Activity.java中注释已经添加的打印输出语句“System.out.println（“the first”）;”，然后在后面添加代码“Log.d(“Log”， “the first Log”);”。

• 运行程序，在DDMS中找到LogCat面板，切换到Log页面，观看打印的内容，如图1-51所示。

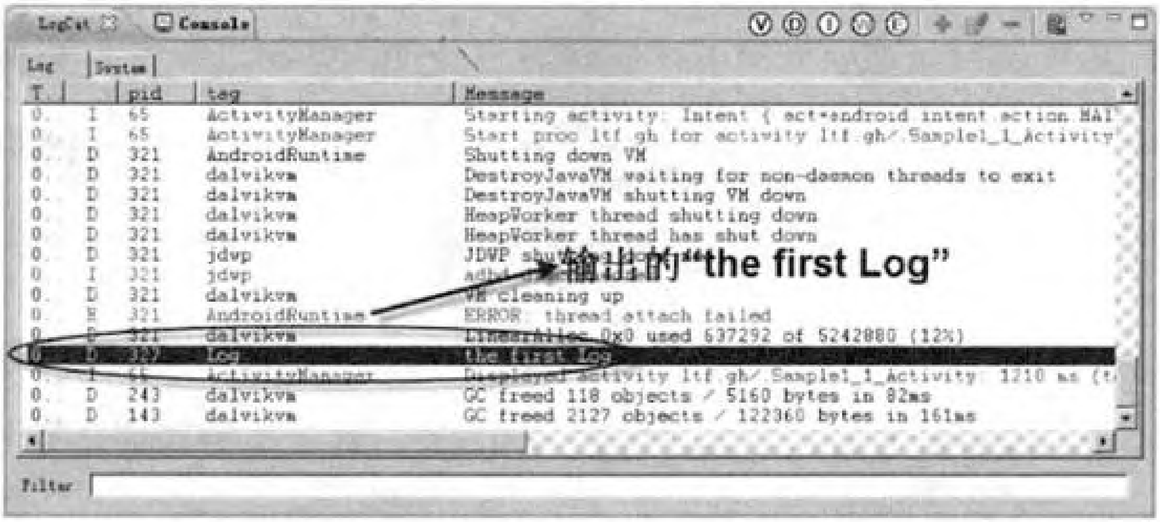


图1-51　使用Log输出测试数据

> **提示**
>
> 使用Log类时需要使用import android.util.Log语句进行导入，System.out.println方法与android.util.Log类各有优缺点，读者可以在开发项目时自行体会，选用自己所需要的。同时需要注意的是，DDMS还有很多强大的功能，这里只介绍了其最基本的用法，有兴趣的读者可以参考2011年9月人民邮电出版社出版的《Android应用案例开发大全》一书的第1.4节“DDMS的灵活应用”或参考其他技术资料。

1.4.4　实际设备的联机调试

由于模拟器在实际项目开发中不是很方便，主要体现为性能较差、很多功能不支持（如本书要重点讨论的OpenGL ES 2.0），因此实际开发中一般使用真机对开发的应用程序进行调试。本小节将介绍实际设备联机调试的方法，基本步骤如下。

• 下载与真机型号相匹配的驱动程序（注意部分手机自带驱动程序，无须下载）。

• 用数据线将手机和开发PC相连接。

• 根据提示，在PC上安装手机驱动程序。

• 将手机设置为允许安装未知来源软件，并打开手机上的USB调试。这两项设置都可以在手机设置选项下的应用程序项目中找到，根据手机具体型号的不同可能略有区别。

• 再次运行前面开发的应用程序，系统将弹出Android Device Chooser窗口，如图1-52所示。

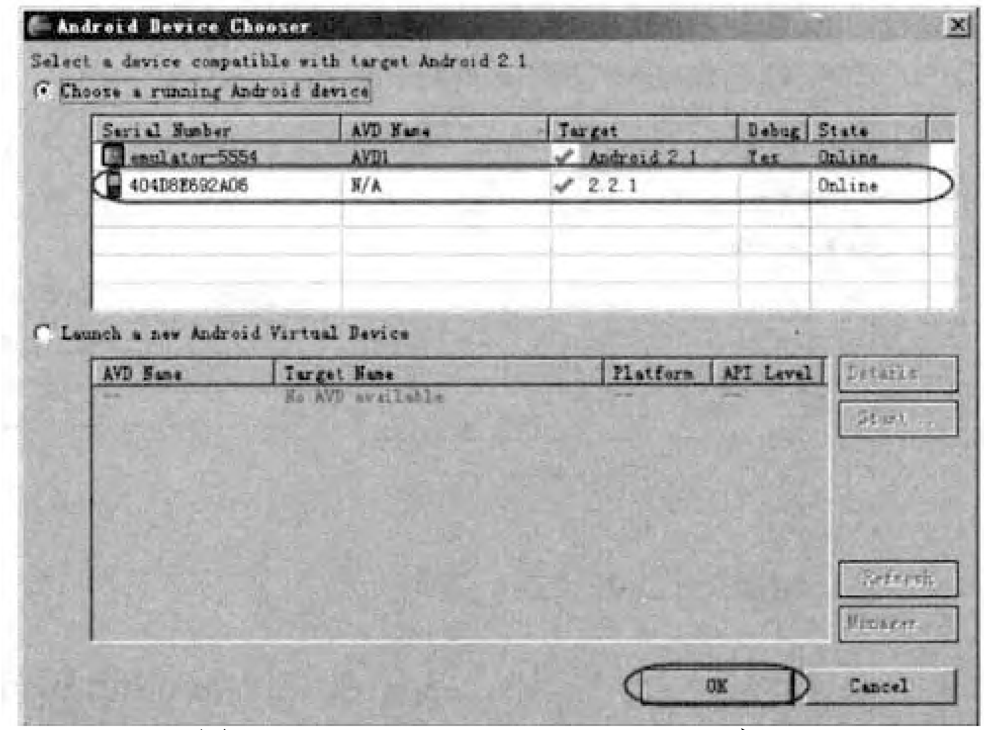


图1-52 Android Device Chooser窗口

• 在窗口中选择自己的设备，然后单击“OK”按钮，程序将送入连接的真机上运行。

• 再打开DDMS，查看左上角的Devices窗口，窗口中列出了真机设备，如图1-53所示。

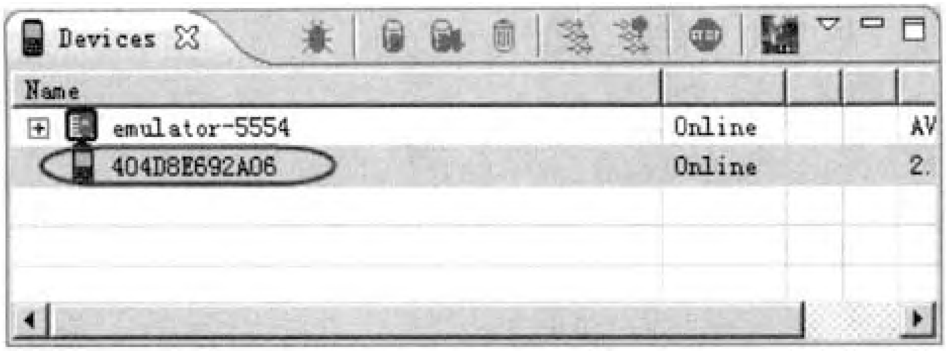


图1-53　DDMS中的Devices窗口

- 选中真机设备，LogCat窗口就出现了真机调试的打印内容，这样就可以在真机中进行调试了。

1.5　Android应用程序运行的机制

上一节介绍了如何搭建Android开发环境、如何开发Hello Android应用程序以及Android应用程序的调试，接下来在本节中将简要地介绍Android应用程序的运行机制。

1.5.1　应用程序的系统架构

Android平台由应用程序、应用程序框架、Android运行时、系统库以及底层Linux内核构成，详细结构如图1-54所示。



图1-54　Android应用程序的系统架构

说明

应用程序层里面包含的就是需要读者去发挥创意开发的、各种各样丰富多彩的应用程序，这也是一般Android应用开发人员工作的层次。

1.5.2　应用程序框架

Android应用程序框架是开发Android应用的基础，开发者在开发时有大部分的时间用在调用这一层。Android应用程序框架包含活动管

理器、窗口管理器、内容提供者、视图系统、包管理器、电话管理器、位置管理器、资源管理器以及通知管理器等，如图1-55所示。



图1-55　Android应用程序框架

应用程序框架层各部分的具体功能如下所列。

- 活动管理器（Activity Manager）：管理所有应用程序的生命周期并提供相应的应用程序导航返回栈。
- 窗口管理器（Window Manager）：管理所有的应用程序窗口。
- 内容提供者（Content Provider）：用来在应用程序之间互相读取或者分享数据。
- 视图系统（View System）：主要包含用来构建应用程序界面的基本控件，包含普通按钮、列表、文本框和单选按钮、复选框等。
- 包管理器（Package Manager）：用来进行Android系统内的程序管理。
- 电话管理器（Telephone Manager）：用来管理所有与移动电话相关的功能。
- 位置管理器（Location Manager）：提供位置服务。
- 资源管理器（Resource Manager）：为应用程序提供各种资源，如图片、布局文件以及本地化字符串等。
- 通知管理器（Notification Manager）：用于管理显示在状态栏的应用程序中的通知信息。状态栏通常在手机的顶部，如短信、语音邮件提示以及剩余电量等全部会在这里显示。

1.5.3　Android运行时

Android运行时包括核心库（Core Libraires）以及Dalvik虚拟机（Dalvik Virtual Machine）两部分，如图1-56所示。



图1-56　Android运行时

- 核心库（Core Libraires）主要包括两部分，一部分是来自于JavaSE中的核心Java部分，如集合框架、多线程等。另一部分由Android核心库如android.os以及android.media等组成。
- Dalvik虚拟机（Dalvik Virtual Machine）是一种基于寄存器的Java虚拟机，主要依靠转换工具dx将Java字节码转换为dex格式（.dex）。基于寄存器的Java虚拟机的优点是所需资源相对较少，同时使用硬件实现虚拟机也会比较容易。

说明

与传统的JavaME不同的是，每个Android应用程序都有一个自己特有的进程，每个应用程序都是使用一个自己持有的Dalvik虚拟机来执行的，而不像在大部分JavaME实现中，多个JavaME应用程序共享一个Java虚拟机进程。

1.5.4　系统库

应用程序框架是最接近于应用程序的软件组件服务，而更底层则是Android的系统函数库（Libraries），这部分是应用程序框架的支撑，其基本架构如图1-57所示。

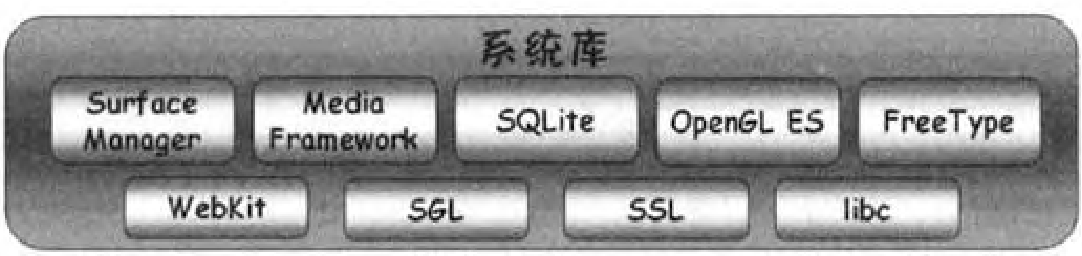


图1-57　系统库

- Surface Manager管理对显示子系统的访问，并且负责将不同应用程序2D及3D绘制的结果进行显示合成。
- Media Framework（媒体函数库）是基于Packet Video公司的OpenCORE发展而成的，使用该函数库可以方便地播放以及录制多种常见格式的影音。这些格式主要包括：MPEG4、H.264、MP3、AAC、AMR、JPG、PNG等。
- SQLite是内嵌到Android系统中的一套轻量级的关系数据库引擎，支持关系数据库的各项功能。
- OpenGL ES：Android是根据OpenGL ES1.0、1.1或2.0的API标准来实现其3D绘图函数

库的，该函数库可以以硬件加速方式执行，也可以用软件模拟方式执行（仅限OpenGL ES1.0与1.1），不过在没有硬件加速的情况下性能会有明显差距。

- FreeType负责点阵字体以及向量字体的绘图显示。
- WebKit本身是一个开源项目，许多浏览器也都是用WebKit引擎所开发的，Android下原生自带的浏览器就是基于其实现的。Android将其引入到系统库，大大方便了开发人员HTTP相关功能的开发。
- SGL库为Android在2D绘图方面提供绘图引擎。
- SSL位于TCP/IP与各种应用层协议之间，为数据通信提供支持。
- Libc提供了一套针对于移动设备优化后的C标准库。

1.5.5　底层Linux内核

Android平台开放性的基础是因为其采用了Linux内核，如图1-58所示。Android平台中的操作系统采用了Linux 2.6版的内核，其包括显示驱动、摄像头驱动、Flash内存驱动、Binder（IPC）驱动、键盘驱动、WIFI驱动、视频驱动及电源驱动等。

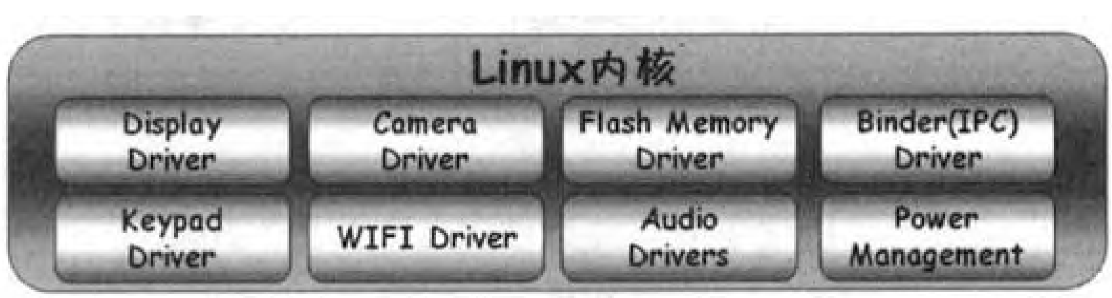


图1-58　底层Linux内核

Linux内核层是软件层与硬件层之间的桥梁，软件开发人员不必关心底层Linux内核的实现细 节，只需将全部精力投入到上层的软件开发中。而实现底层Linux内核的工作是由Google和手机厂商来完成，如驱动的更新或者新驱动的编写等。

1.6 本章小结

本章主要分为5个部分，分别介绍了手机市场的现状、Android平台的特点及未来的趋势、如何搭建Android开发环境、如何开发Hello Android应用程序、如何调试Android应用程序以及Android应用程序的运行机制。通过阅读本章，读者应该对当下的手机产业有个大概的了解，初步掌握Android相关的环境变量配置，简单程序的开发与调试，并且对Android应用程序的运行机制有大概的了解。

第2章　游戏开发相关Android基础知识

想要进行复杂游戏的开发，首先需要掌握与游戏相关的很多Android基础知识。本章将会详细地向读者介绍在Android游戏开发中必备的一些基础知识，如音效、文件的读取、游戏信息的存储、Socket网络、蓝牙网络等。

> **提示**
>
> 由于本书侧重于Android下3D高级开发方面的内容，故在进行本章案例的介绍时，主要是侧重各项具体功能技术的介绍，对于界面控件部分的开发细节省略，需要学习的读者可以参考光盘中界面设计的xml源代码。若对Android基础知识很不熟悉的读者，可以参考笔者在人民邮电出版社出版的《Android应用开发完全自学手册——核心技术、传感器、2D/3D、多媒体与典型案例》一书，其中详细介绍了Android各方面的基础知识。

2.1　游戏中的音效

一款好游戏，除了具备优质的画面和较高的可玩性之外，还应该有出色的音效。音效一般指的是游戏中发生特定行为或进行特定操作时播放的效果音乐或为了渲染整体气氛播放的背景音，如远处隆隆的炮声、怪物死亡的惨叫声、由远而近的脚步声等。

通过开发人员精心准备的声音特效，结合游戏的场景，可以渲染出一种紧张刺激的氛围，使玩家产生身临其境的感觉。这就像电影中的声音特效一样，假如没有了合适的音效，那么游戏和电影一样，真实感会大打折扣。

> **提示**
>
> 按照作用的不同，可以将音效划分为即时音效和背景音乐。两种音效在Android中的实现技术是不同的，本节将向读者详细介绍两种音效在Android中的具体实现。

2.1.1　游戏中的即时音效

游戏中有时需要根据情况播放即时音效，如枪炮声、碰撞声等。即时音效的特点是短暂、可以重复、可以同时播放。由于Android提供的MediaPlayer（媒体播放器）会占用大量的系统资源，而且播放时还需要进行缓冲，有较大的时延，因此使用MediaPlayer无法实现即时音效。

Android系统的设计者也考虑到了这个问题，为即时音效的实现提供了一个专门的类—— SoundPool。SoundPool类用于管理和播放应用程序中的声音资源，使用该类时首先需要通过该类将声音资源加载到内存中，然后在需要即时音效的地方播放即可，其几乎没有时延，可以满足游戏实时性的需要。

> **提示**
>
> 由于SoundPool设计的初衷是用于无时延地播放游戏中的短促音效，因此实际开发中应该只将长度小于7s的声音资源放进SoundPool，否则可能加载失败或内存占用过大。

SoundPool类的构造器及常用方法如表2-1所列。

表2-1 SoundPool类的构造器以及方法

构造器或方法签名	方法含义	各个参数含义
public SoundPool (int maxStreams, int streamType,int srcQuality)	创建SoundPool对象	maxStreams：该参数用于设置最多同时能够播放多少个音效 streamType：该参数设置音频类型，在游戏中通常设置为：STREAM_MUSIC srcQuality：该参数设置音频文件的质量，目前还没有具体作用，设置为0（默认值）
public int load(Context context, int resId, int priority)	加载音频文件	context：应用程序的上下文 resId：该参数为要加载的音效资源的ID priority：优先级，现在还没有作用，设置为1即可
public final int play (int soundID, float leftVolume, float rightVolume, int priority, int loop, float rate)	播放音效	soundID：该参数为要播放音效的ID leftVolume：该参数用来控制左声道音量 rightVolume：该参数用来控制右声道音量 priority：该参数为优先级，0为最低 loop：该参数为音效播放的循环次数，0为不循环，-1为永远循环 rate：该参数为音效的回放速度，该值在0.5-2.0f，1.0f为正常速度
public final void pause (int streamID)	暂停音效的播放	streamID：要暂停音效的ID
public final void stop(int streamID)	停止音效的播放	streamID：要停止音效的ID

2.1.2 即时音效的一个案例

了解了SoundPool类的基本操作方法之后，接下来就可以开发游戏中用到的即时音效了。本小节将向读者展示一个播放和停止即时音效的简单案例，其主要功能为，通过SoundPool声音池技术来实现一个即时音效的播放和停止，运行效果如图2-1和图2-2所示。

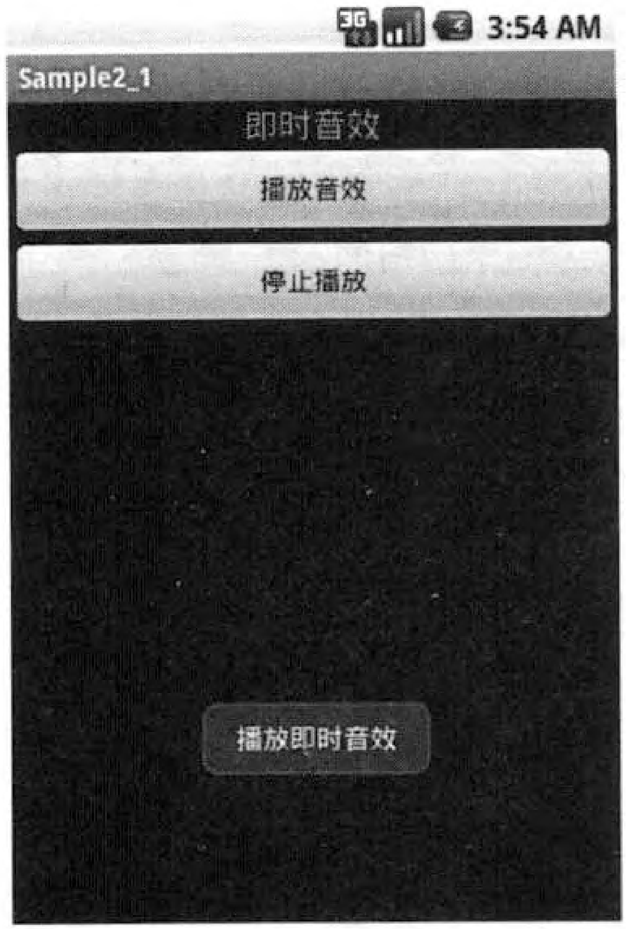


图2-1　播放即时音效

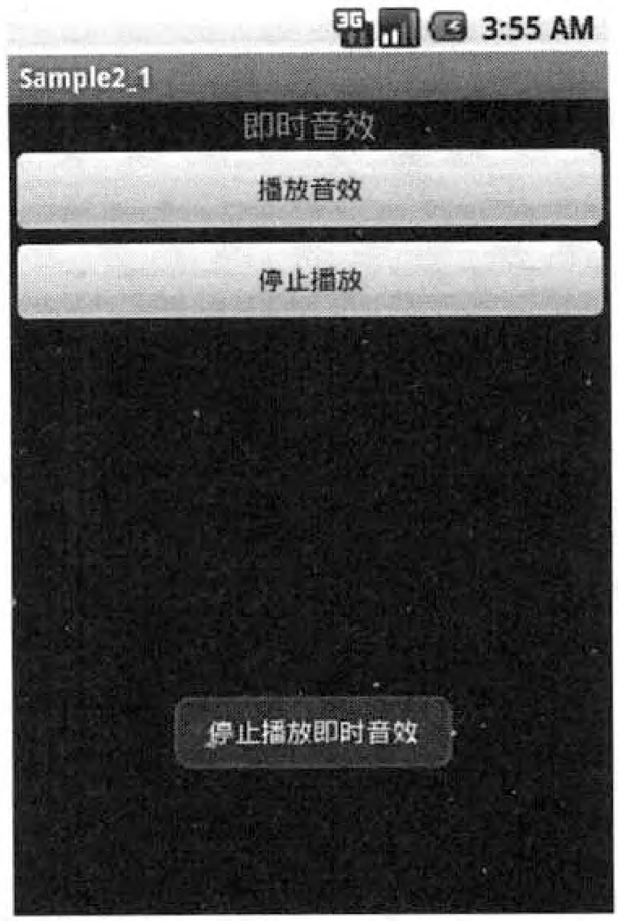


图2-2　停止播放音效

了解了本案例的运行效果后，接下来对其具体开发步骤进行介绍，具体如下所列。

（1）首先在Eclipse中新建名称为Sample2_1的项目，然后在项目目录下的res文件夹下新建raw文件夹。接着将需要被播放的短促音效对应的音频文件musictest.ogg复制到raw文件夹下，如图2-3所示。



图2-3　导入音效文件

> **提示**
>
> 一般在Android手机平台上使用的即时音效文件越小越好，这有助于提高游戏的整体性能。对于同一个音效文件，在不改变其时长的情况下，可以采用降低采样率（如降低到16Kbit/s）或由立体声改为单声道的方式来缩小体积。

（2）准备好声音资源后，接下来进行本案例中Sample2_1_Activity类的开发。该类中使用了声音池技术实现了即时音效的播放，其代码如下。

代码位置：见随书光盘中源代码/第2章/Sample2_1/src/com/bn/pp1目录下的Sample2_1_Activity.java。

```
package com/bn/ppl;                                    //声明包
import java.util.HashMap;                              //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.widget.Toast;                           //引入相关类
public class Sample2_1_Activity extends Activity {
    SoundPool sp;                                      // 声明SoundPool的引用
    HashMap<Integer, Integer> hm;                      // 声明HashMap来存放声音文件
    int currStreamId;                                  // 当前正播放的streamId
    @Override
                                                       // 重写onCreate方法
        public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);             // 跳转到主界面
            initSoundPool();                           // 初始化声音池的方法
            Button b1 = (Button) this.findViewById(R.id.Button01); // 获取播放按钮
            b1.setOnClickListener                      // 为播放按钮添加监听器
            (new OnClickListener() {
                @Override
                public void onClick(View v) {
                    playSound(1, 0);                   // 播放1号声音资源，且播放一次
                                                       // 提示播放即时音效
                    Toast.makeText(getBaseContext(), "播放即时音效", Toast.LENGTH_SHORT)
```

```
23                          .show();
24                  }
25          });
26          Button b2 = (Button) this.findViewById(R.id.Button02); // 获取停止按钮
27          b2.setOnClickListener                       // 为停止按钮添加监听器
28          (new OnClickListener() {
29              @Override
30              public void onClick(View v) {
31                  sp.stop(currStreamId);              // 停止正在播放的某个声音
32                                                      // 提示停止播放
33                  Toast.makeText(getBaseContext(), "停止播放即时音效",
34                          Toast.LENGTH_SHORT)
35                          .show();
36              }
37          });}
38                                                      // 初始化声音池的方法
39      public void initSoundPool() {
40          sp = new SoundPool(4, AudioManager.STREAM_MUSIC, 0); // 创建SoundPool对象
41          hm = new HashMap<Integer, Integer>();       // 创建HashMap对象
42                                      // 加载声音文件musictest并且设置为1号声音放入hm中
43          hm.put(1, sp.load(this, R.raw.musictest, 1));
44      }
45                                                      // 播放声音的方法
46      public void playSound(int sound, int loop) { // 获取AudioManager引用
47          AudioManager am = (AudioManager) this
48                  .getSystemService(Context.AUDIO_SERVICE);
49                                                      // 获取当前音量
50          float streamVolumeCurrent = am
51                  .getStreamVolume(AudioManager.STREAM_MUSIC);
52                                                      // 获取系统最大音量
53          float streamVolumeMax = am
54                  .getStreamMaxVolume(AudioManager.STREAM_MUSIC);
55                                                      // 计算得到播放音量
56          float volume = streamVolumeCurrent / streamVolumeMax;
57                                          // 调用SoundPool的play方法来播放声音文件
58          currStreamId = sp.play(hm.get(sound), volume, volume, 1, loop, 1.0f);
59  }}
```

- 第6-8行为声明所用到的SoundPool和HashMap对象的引用，其中SoundPool用来加载、播放、停止音效；HashMap用来管理音效id。currStreamId为当前正播放的streamId，用于对正在播放的音效进行管理。
- 第14行调用initSoundPool方法来对声音池进行初始化。
- 第15-37行为播放和停止按钮添加监听器，并在单击操作时显示Toast进行提示。
- 第39-44行为初始化声音池的方法，其首先创建SoundPool对象，然后加载音效文件到声音池并将生成的音效id存储进HashMap中。

• 第46-59行为播放声音的方法，其中调用SoundPool的play方法来实现即时音效的播放。

> **提示**
>
> 通过以上的案例，读者可以看出使用SoundPool播放即时音效是非常简单的。今后的游戏开发中，只要是游戏中的即时音效都应该用此方式来实现。

2.1.3 背景音乐播放技术

背景音乐也可以采用前一小节的声音池技术，在播放背景音乐的时候，只需要把loop播放次数参数设置成-1进行无限循环即可。但由于SoundPool只适合播放不大于7秒的音效文件，限制较大。而背景音乐对时延并不敏感，因此在实际的游戏开发中，时长较长的背景音乐一般采用媒体播放器（MediaPlayer）来进行播放。

要想很好地使用MediaPlayer进行音/视频文件的播放，首先必须要熟悉MediaPlayer的生命周期。这样不仅有利于开发人员开发出更加合理的代码，而且可以达到充分利用系统资源的目的。

1. MediaPlayer的生命周期

MediaPlayer的生命周期包括10种状态，每种状态下可以调用相应的方法来实现音/视频文件的管理或播放。其各个状态及状态间的关系可以用一个简单的流程图来表示，如图2-4所示。

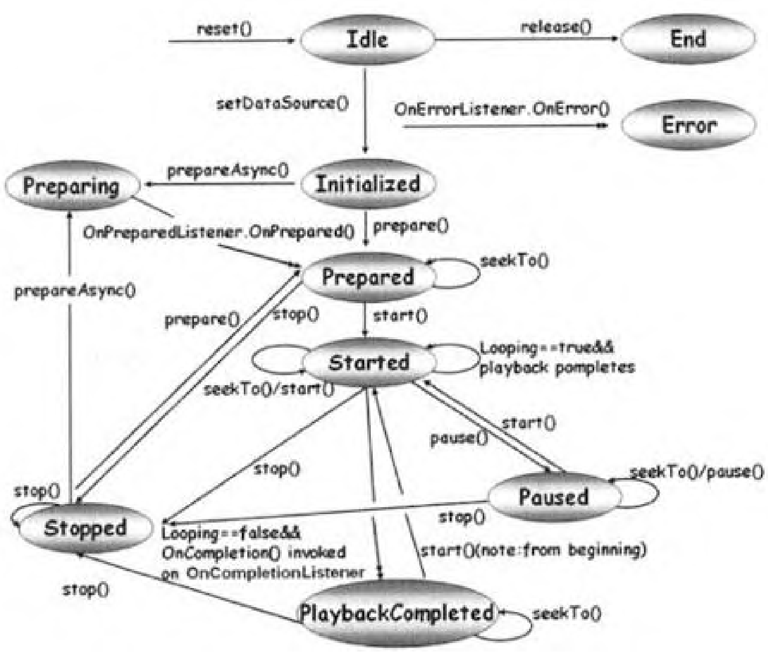


图2-4 MediaPlayer的生命周期图

- Idle状态。

使用new方法创建一个MediaPlayer对象或者调用了其reset方法时，该MediaPlayer对象处于idle状态。

但通过两种不同方式进入的idle状态还是有些区别的，主要体现为：如果在这个状态下调用了getDuration等方法，若是通过reset方法进入idle状态的话会触发OnErrorListener.onError，并且MediaPlayer会进入Error状态；如果是新创建的MediaPlayer对象，则并不会触发onError，也不会进入Error状态。

- End状态。

通过release方法可以进入End状态，只要MediaPlayer对象不再被使用，就应当尽快将其通过release方法释放掉，以释放其占用的软、硬件资源，这其中有些资源是互斥的（相当于临界资源）。如果MediaPlayer对象进入了End状态，则不会再进入其他任何状态了。

- Initialized状态。

这个状态比较简单，MediaPlayer调用setDataSource方法就进入了Initialized状态，表示此时要播放的文件已经设置好了。

- Prepared状态。

初始化完成之后还需要通过调用prepare或prepareAsync方法进行准备，这两个方法一个是同步的，一个是异步的。只有进入了Prepared状态，才表明MediaPlayer到目前为止都工作正常，可以进行音乐文件的播放。

- Preparing状态。

这个状态比较容易理解，主要是与prepareAsync异步准备方法配合，如果异步准备完成，会触发OnPreparedListener.onPrepared，进而进入Prepared状态。

- Started状态。

MediaPlayer准备完成后，通过调用start方法，将进入Started状态。所谓Started状态，也就是播放中状态，开发中可以使用isPlaying方法测试MediaPlayer是否处于Started状态。

如果播放完毕，而又设置了循环播放，则MediaPlayer仍然会处于Started状态。类似的，如果在该状态下MediaPlayer调用了seekTo或者start方法均可以让MediaPlayer停留在Started状态。

- Paused状态。

Started状态下调用pause方法可以暂停播放，从而进入Paused状态。MediaPlayer暂停后再次调用start方法则可以继续进行播放，并转到Started状态。暂停状态时可以调用seekTo方法，这是不会改变状态的。

- Stop状态。

Started或Paused状态下均可调用stop方法停止播放并进入Stop状态，而处于Stop状态的MediaPlayer要想重新播放，需要通过调用prepareAsync或prepare方法返回到先前的Prepared状态重新开始才可以。

- PlaybackCompleted状态。

文件正常播放完毕，而又没有设置循环播放的话就进入该状态，并会触发OnCompletionListener接口中的onCompletion方法。此时可以调用start方法重新从头播放文件，也可以调用stop方法停止播放，或者调用seekTo方法来重新定位播放位置。

- Error状态。

由于某种原因MediaPlayer出现了错误，则会触发OnErrorListener.onError回调方法，此时MediaPlayer即进入Error状态。及时捕捉并妥善处理这些错误是很重要的，这可以帮助应用程序及时释放相关的软、硬件资源，也可以改善用户体验。

如果MediaPlayer进入了Error状态，可以通过调用reset方法来恢复，使得MediaPlayer重新返回到Idle状态。

> **提示**
>
> 从上述对生命周期的介绍中可以看出，某些情况发生时MediaPlayer会回调特定监听接口中的事件处理方法。若读者在开发中希望使用回调，则需要首先向MediaPlayer注册实现了指定监听接口的监听器。例如，可以使用setOnErrorListener方法注册实现了OnErrorListener接口的监听器，当MediaPlayer进入Error状态时监听器中的onError方法就会被回调。

2．AudioManager类

AudioManager类在Android系统中主要用来进行音/视频播放时的音量控制，使用时的基本步骤如下所列。

- 首先可以调用Activity对象的getSystemService（Context.AUDIO_SERVICE）方法获取AudioManager对象。
- 然后再调用AudioManager类中的相关方法进行音量控制。

AudioManager类中的常用方法如表2-2所列。

表2-2　AudioManager类中的常用方法

方　法　名	说　明
public void adjustVolume (int direction, int flags)	调用该方法可以调节音量大小。参数direction为调节的方向，ADJUST_LOWER减小音量、ADJUST_RAISE增大音量、ADJUST_SAME保持音量不变；参数flags为标记，通常设置为0
public int getStreamMaxVolume (int streamType)	调用该方法可以得到系统最大音量
public int getStreamVolume (int streamType)	调用该方法可以得到系统当前音量
void android.media.AudioManager.setStreamVolume(int streamType, int index, int flags)	调用该方法可以设置音量的大小，参数streamType为声音的类型，index为要设置的音量的索引值，参数flags为标记，通常设置为AudioManager.FLAG_PLAY_SOUND

> **提示**
>
> MediaPlayer类还可以对视频文件进行操作，由于本书只介绍与游戏音效相关的功能，因此不再对其进行介绍，有兴趣的读者可以自行查阅相关资料。

2.1.4 简易音乐播放器的实现

了解了MediaPlayer和AudioManager类的基本操作方法之后，就可以对游戏的背景音乐功能进行开发了。本小节将通过这两个类来实现一个简易的音乐播放器，其主要功能为对手机SD卡中的音乐文件进行播放，运行效果如图2-5所示。



图2-5　案例运行效果图

> **提示**
>
> 图2-5中从左到右分别为案例运行后，依次单击“播放音乐”按钮、“暂停播放”按钮、“增大音量”按钮后的效果图。停止播放音乐和减小音量的效果与图2-5中的已有效果类似，这里没有给出，请读者自行运行本案例进行查看。

了解了案例的具体运行效果后，接下来就介绍案例的具体开发步骤，具体如下所列。

（1）首先需要准备好要播放的音乐文件，本案例中使用的是著名的高山流水古曲，文件名为 “gsls.mp3”。准备完音乐文件后，将该音乐文件通过DDMS导入到模拟器或真机的SD卡中，如图2-6所示。

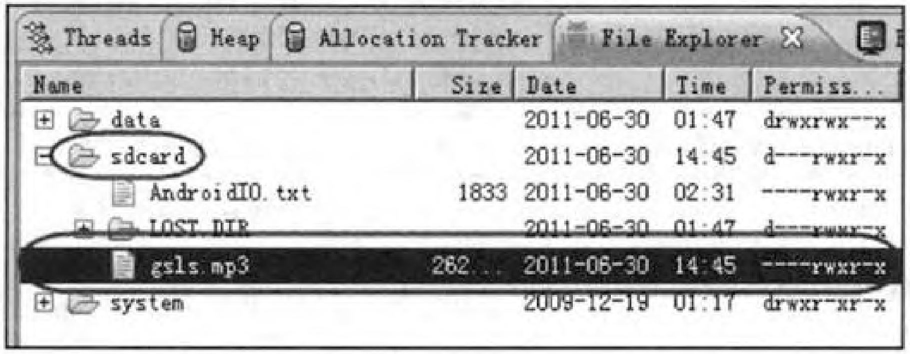


图2-6　导入音乐文件

提示

高山流水曲的音乐资源文件见随书光盘中源代码/第2章目录下的gsls.mp3。导入时直接用鼠标光标将文件拖曳到DDMS下“File Explorer”面板中的“sdcard”目录下即可。

（2）音乐资源在SD卡中放置完成后，接下来进行本案例中Sample2_2_Activity类的开发。该类使用了MediaPlayer实现了背景音乐的播放，具体代码如下。

代码位置：见随书光盘中源代码/第2章/Sample2_2/src/com/bn/pp2目录下的Sample2_2_Activity.java。

```
package com.bn.pp2;                                   //声明包
import java.io.IOException;                           //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.widget.Toast;                          //引入相关类
public class Sample2_2_Activity extends Activity {
    MediaPlayer mp;                                   // 声明MediaPlayer的引用
```

```
AudioManager am;                                      // 声明 AudioManager 的引用
private int maxVolume;                                // 最大音量值
private int currVolume;                               // 当前音量值
    private int stepVolume;                           // 每次调整的音量幅度
    @Override
    public void onCreate(Bundle savedInstanceState) // 重写 onCreate 方法
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);                // 跳转到主界面
        mp = new MediaPlayer();          // 创建 MediaPlayer 实例对象
        try {
            mp.setDataSource("/sdcard/gsls.mp3"); //为 MediaPlayer 设置要播放文件资源
            mp.prepare();                // MediaPlayer 进行缓冲准备
        }
        catch (Exception e) {
            e.printStackTrace();
        }
                                         // 获取 AudioManager 对象引用
        am = (AudioManager) this.getSystemService(Context.AUDIO_SERVICE);
                                         // 获取最大音乐音量
        maxVolume = am.getStreamMaxVolume(AudioManager.STREAM_MUSIC);
                                         // 每次调整的音量大概为最大音量的 1/6
        stepVolume = maxVolume / 6;
        Button bstart = (Button) this.findViewById(R.id.Button01); // 获取开始按钮
        bstart.setOnClickListener        // 为开始按钮添加监听器
        (new OnClickListener() {
            @Override
            public void onClick(View v) {
                mp.start();              // 调用 MediaPlayer 的 start 方法来播放音乐
                Toast.makeText(getBaseContext(), "开始播放 '高山流水曲'",
                        Toast.LENGTH_LONG).show();
            }
        });
        Button bpause = (Button) this.findViewById(R.id.Button02); // 获取暂停按钮
        bpause.setOnClickListener        // 为暂停按钮添加监听器
        (new OnClickListener() {
            @Override
            public void onClick(View v) {
                mp.pause();              // 调用 MediaPlayer 的 pause 方法暂停播放音乐
                Toast.makeText(getBaseContext(), "暂停播放'高山流水曲'",
                        Toast.LENGTH_LONG).show();
            }
        });
        Button bstop = (Button) this.findViewById(R.id.Button03); // 获取停止按钮
        bstop.setOnClickListener                             // 为停止按钮添加监听器
        (new OnClickListener() {
            @Override
            public void onClick(View v) {
                mp.stop();               // 调用 MediaPlayer 的 stop 方法停止播放音乐
                try {
                    mp.prepare();                     //进入准备状态
```

```
58                } catch (IllegalStateException e) {    //捕获异常
59                    e.printStackTrace();
60                } catch (IOException e) {              //捕获异常
61                    e.printStackTrace();
62                }
63                Toast.makeText(getBaseContext(), "停止播放'高山流水曲'",
64                        Toast.LENGTH_LONG).show();
65            }
66        });
67        Button bUp = (Button) this.findViewById(R.id.Button04); // 获取增大音量按钮
68        bUp.setOnClickListener                     // 为增大音量按钮添加监听器
69        (new OnClickListener() {
70            @Override
71            public void onClick(View v) {
72                                                   // 获取当前音量
73                currVolume = am.getStreamVolume(AudioManager.STREAM_MUSIC);
74                                                   //增加音量，但不超过最大音量值
75                int tmpVolume = currVolume + stepVolume;    //临时音量
76                currVolume = tmpVolume < maxVolume ? tmpVolume:maxVolume;
77                am.setStreamVolume(AudioManager.STREAM_MUSIC, currVolume,
78                        AudioManager.FLAG_PLAY_SOUND);
79                Toast.makeText(getBaseContext(), "增大音量",
80                        Toast.LENGTH_SHORT).show();
81            }
82        });
83        Button bDown = (Button) this.findViewById(R.id.Button05); // 获取减小音量按钮
84        bDown.setOnClickListener                       // 为减小音量按钮添加监听器
85        (new OnClickListener() {
86            @Override
87            public void onClick(View v) {
88                                                       // 获取当前音量
89                currVolume = am.getStreamVolume(AudioManager.STREAM_MUSIC);
90                                                       //减小音量，但不小于0
91                int tmpVolume = currVolume - stepVolume;    //临时音量
92                currVolume = tmpVolume > 0 ? tmpVolume:0;
93                am.setStreamVolume(AudioManager.STREAM_MUSIC, currVolume,
94                        AudioManager.FLAG_PLAY_SOUND);
95                Toast.makeText(getBaseContext(), "减小音量",
96                        Toast.LENGTH_SHORT).show();
97            }
98    });}}
```

- 第6-10行为声明需要用到的MediaPlayer和AudioManager对象的引用、最大音量值、当前音量值及每次调整的音量幅度等。
- 第16-23行获取了MediaPlayer类对象的引用，设置了音乐资源的路径，并调用prepare方法进入了准备状态。
- 第24-29行获取了AudioManager类对象的引用，同时获取了音乐播放的最大音量值，并设置每次调整的音量幅度为最大值的1/6。
- 第30-98行分别获取了开始按钮、暂停按钮、停止按钮以及增大和减小音量按钮对象的引用， 并分别为这些按钮添加了监听器。这样在用户单击这些按钮时程序就

能完成指定的功能了，例如，按下开始按钮开始播放音乐、按下减小音量按钮减小播放音量等。

2.2 简单数据的存储——Preferences

进行游戏开发时，有时需要持久地存储一些十分简单的数据，例如，游戏中音效的开关状态、振动效果的开关状态等。Android设计者也充分注意到了这种存储简单数据的需求，提供了一个Preferences类专门用于完成此类任务。

2.2.1 Preferences简介

Preferences使用非常方便，但功能也不是很强大，一般适合用来保存一些用户设置的参数，是一种轻量级的数据存储机制。Preferences仅可以用来存储几种简单类型的数据，如：boolean、int、float、long或者String。这些数据以键值对的形式存储在应用程序私有的Preferences目录（/data/data/应用程序包名/shared_prefs/）下的特定xml文件中，如图2-7所示。

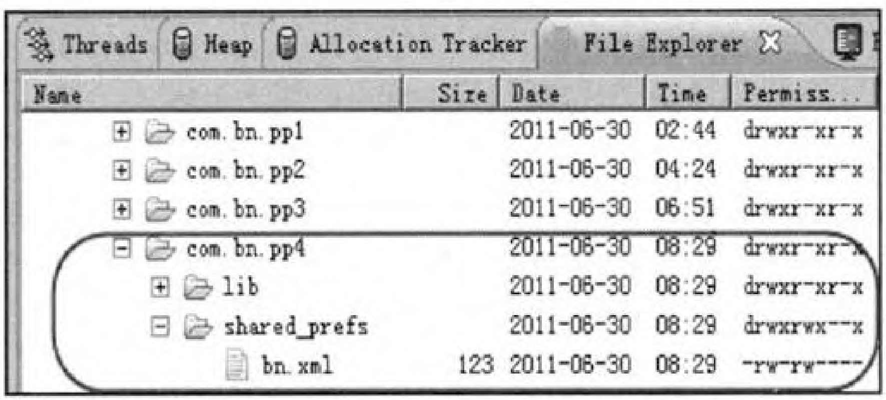


图2-7　Preferences文件存放的位置

> **提示**
>
> 请读者注意，这个xml文件是设计为由程序自动维护的，不应该手动人工进行修改，以免影响程序的正确运行。

2.2.2 Preferences实现访问时间的记录

本小节将介绍一个使用Preferences实现的，具有记录上次访问时间功能的小案例。其具体功能为：第一次运行本案例中的程序，程序将显示“用户您好，欢迎您第一次光临本软件。”，若不是第一次运行，则显示上一次用户运行的时间，如“用户您好，您上次进入时间为：Oct 15, 2011 2:38:40 AM”。程序运行的效果如图2-8、图2-9所示。

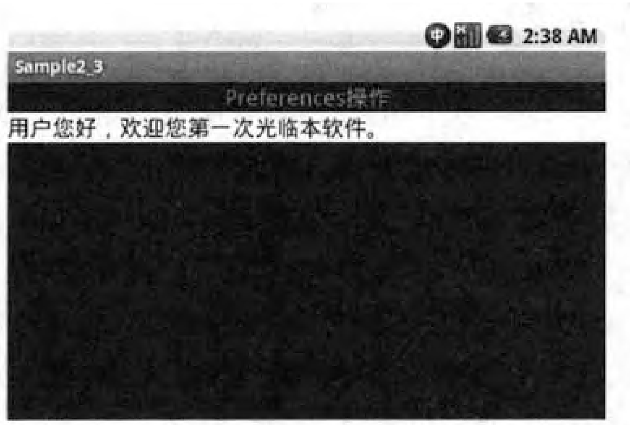


图2-8　第一次打开本案例

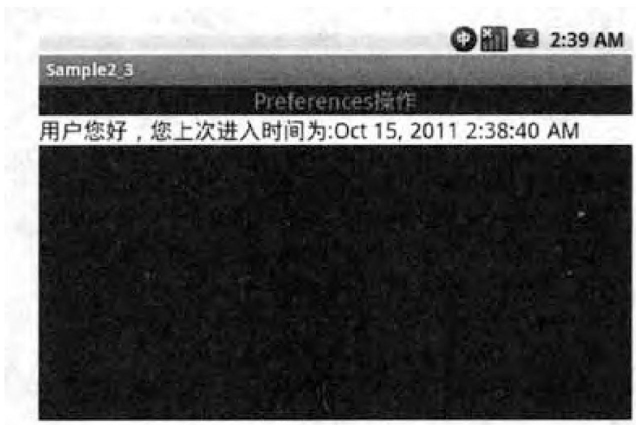


图2-9　再次打开本案例

介绍完本案例的运行效果之后，接下来将对本案例的开发进行介绍。主要是本案例的主控制类Sample2_3_Activity的开发，其代码如下。

代码位置：见随书光盘中源代码/第2章/Sample2_3/src/com/bn/pp3目录下的Sample2_3_Activity.java。

```
package com/bn/pp3;
import java.util.Date;                                   //引入相关包
public class Sample2_3_Activity extends Activity {       //创建Activity
```

```
4    @Override
5    public void onCreate(Bundle savedInstanceState) {//重写 onCreate 方法
6        super.onCreate(savedInstanceState);
7        setContentView(R.layout.main);              //跳转到主界面
8        //获取 SharedPreferences 引用，存储名称为 bn.xml，读写模式为私有
9        SharedPreferences sp=this.getSharedPreferences("bn", Context.MODE_PRIVATE);
10           String lastLoginTime=sp.getString("time", null);  //获取键为 "time" 的值
11           if(lastLoginTime==null) {               //若值为空，则为第一次运行本案例
12             lastLoginTime="用户您好，欢迎您第一次光临本案例。";    }
13           else{                                   //不为空，则修改字符串为上次运行时间
14                 lastLoginTime="用户您好，您上次进入时间为:"+lastLoginTime;    }
15          SharedPreferences.Editor editor=sp.edit();  //取得编辑对象，来修改 Preferences 文件
16           editor.putString("time", new Date().toLocaleString());
                                                     //修改键为 "time" 的值为当前时间
17           editor.commit();                        //提交修改
18           TextView tv=(TextView)this.findViewById(R.id.TextView01); //获取用来显示的
                                                                       //TextView
19           tv.setText(lastLoginTime);              //设置显示的字符串
20    }}
```

- 第9行使用getPreferences0方法来获得Preferences对象，并将存储文件命名为bn.xml，同时规定读写模式为私有。
- 第10-14行功能为在打开程序时，获取Preferences中存储的键为“time”的值，并判断是否为空来确定是否为第一次运行本案例，来显示不同的信息。
- 第15-17行功能为首先获取Preferences的编辑对象，然后对Preferences文件进行修改，设置键为“time”的值为当前的时间，最后提交。
- 第18-19行功能为设置界面中显示的字符串内容。

2.3 手机自带数据库——SQLite

上一节介绍了如何使用Preferences存储简单数据，而复杂的数据就需要存储到文件或数据库中了。Android自带了一款轻量级的关系数据库——SQLite，其具有体积小，功能强大等诸多特点， 成为嵌入式设备首选的数据库系统。本节将带领读者走进SQLite的世界，去学习如何应用SQLite数据库进行数据的增、删、改、查等基本操作。

2.3.1 初识SQLite

SQLite是一款满足ACID特性的具有完备功能的关系数据库系统，由于其设计目标为轻量级、开源、支持嵌入式使用，因此，目前已经在嵌入式设备领域被广泛采用。其运行需要的系统资源非常少，在嵌入式设备中可能只需要几百KB的内存就够了。

SQLite对主流编程语言的支持也非常全面，如C#、PHP、Java等，同时还支持ODBC接口。另外，SQLite的性能也是一流的，在一般应用情况下，其处理速度比MySQL、PostgreSQL这两款著名的开源数据库管理系统都快。

> **提示**
>
> SQLite的最新版本为3.7.6，发布时间是2011年4月12日。其官方网站为：http://www.sqlite.org或者http://www.sqlite.com.cn，读者可以在该网站上获取SQLite的源代码和相关文档。

虽然SQLite占用的资源非常少，但是其功能、特性与服务器级数据库相比却丝毫不差，这也是SQLite能够受到Android系统青睐的主要原因，其部分特性如下所列。

- 最大可以支持2TB的数据库文件。
- 占用资源少，一般占用250KB左右。
- API非常简单，易于使用。
- 没有任何额外的依赖，是独立的。
- 源代码完全开放，可以用于任何用途。

Android系统中很多的用户数据都是存储在SQLite数据库中的，如联系人信息、通话记录、短信等，由此可见SQLite对于Android的重要性。

> **提示**
>
> 读者要想很好地使用SQLite数据库，必须熟练掌握SQL语言。这是由于SQL已经事实上成为关系数据库操作的标准语言，市面上的关系数据库几乎无一例外都支持SQL。因此在数据库领域，有这样一句话“学好SQL，走遍天下都不怕”。

2.3.2 SQLite数据库的基本操作

一般学习数据库相关课程的时候，首先介绍的就是数据库的一些基本操作，如数据的增、删、改、查等。按照惯例，本书也首先简单介绍SQLite数据库的创建、关闭及数据的增加、删除、修改、查询等基本操作，具体如下所列。

> **提示**

想在Android下通过Java编程对SQLite数据库进行操作，就必须要用到android.database.sqlite包下的SQLiteDatabase类，该类提供了对SQLite数据库进行基本操作的所有重要方法。

- 创建数据库。

创建数据库需要用到的是openDatabase方法，此方法签名为“public static SQLiteDatabase openDatabase（String path, SQLiteDatabase.CursorFactory factory, int flags）”。其中path为数据库所在的路径；factory为游标工厂；flags为标志，可以用来控制数据库的访问模式。

- 关闭数据库。

关闭数据库需要用到的是close方法，此方法签名为“public void close（）”。在实际开发中数据库使用完毕后，一定不要忘记使用该方法关闭数据库，以防止资源的浪费。

- 插入数据。

插入数据可以使用insert方法，此方法签名为“public long insert（String table, String nullColumnHack, ContentValues values）”。其中table为待插入的表名，nullColumnHack通常设置为null，values为待插入的数据。

- 更新数据。

更新数据可以使用update方法，其签名为“public int update（String table, ContentValues values，String whereClause, String[] whereArgs）”。其中table为待更新的表名；values为待更新内容；whereClause为where子句的内容，用来进行记录筛选；whereArgs为where子句的参数。

- 删除数据。

删除数据可以使用delete方法，其签名为“public int delete（String table, String whereClause, String[] whereArgs）”。其中table为要操作的表名；whereClause为where子句的内容，用来进行记录筛选；whereArgs为where子句的参数。

- 查询数据。

查询数据可以使用query方法，其方法签名为“public Cursor query（String table, String[] columns，String selection, String[] selectionArgs, String groupBy, String having, String orderBy）”。其中table为要查询的表，columns为要查询的列，selection为过滤记录的子句，selectionArgs为过滤的参数值，groupBy为分组子句，having为过滤分组的子句，orderBy为记录排序子句。

提示

Android中query方法被重载了，有多个变体。这里由于篇幅所限，不再赘述，有需要的读者可以自行查阅API或其他相关资料。

- 执行非查询SQL。

对于不太熟悉SQL语言的初学者而言，插入、更新、删除数据可以用前面介绍的insert、update、delete方法。但对于熟练掌握SQL的开发人员而言，就是使用execSQL方法直接执行相应的SQL语句更方便了。

此方法签名为“public void execSQL（String sql）”或“public void execSQL（String sql，Object[] bindArgs）”。其中sql为需要执行的SQL语句，bindArgs为带参数SQL语句的参数值数组。

> **提示**
>
> 需要注意的是，此方法仅支持执行非查询的SQL语句，如CREATE TABLE、DELETE、INSERT、UPDATE等，不能用于执行SELECT语句。

- 执行查询SQL。

对于熟练掌握SQL的开发人员而言，会觉得前面介绍的query方法使用过于烦琐。Android的设计人员也考虑到了这个问题，提供了支持执行查询SQL语句的rawQuery方法，其方法签名为“public Cursor rawQuery（String sql，String[] selectionArgs）”。其中sql为要执行的查询SQL语句（可以带参数），selectionArgs为查询参数的值。

> **提示**
>
> SQLiteDatabase类中用于数据库操作的方法还有很多，本书只是介绍了其中一些常用的，若读者有需要可以查阅API或其他相关资料进一步学习。

2.3.3 SQLite数据库的简单案例

上一小节介绍了SQLite数据库的基本操作方法，本小节将详细介绍一个使用SQLite数据库的简单案例，以使读者可以更加快速地掌握SQLite数据库的使用方法，从而在开发中进行合理地使用。本案例运行效果分别如图2-10、图2-11和图2-12所示。

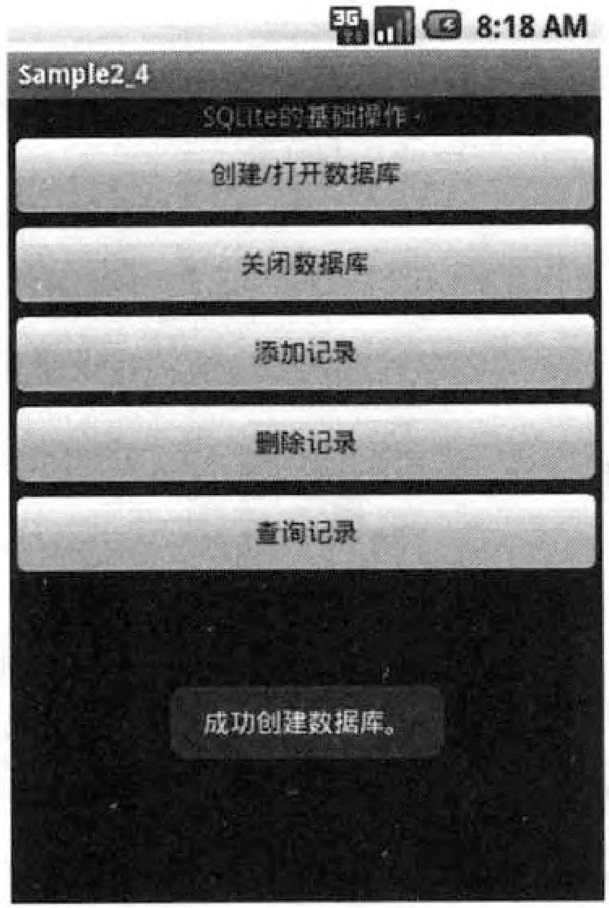
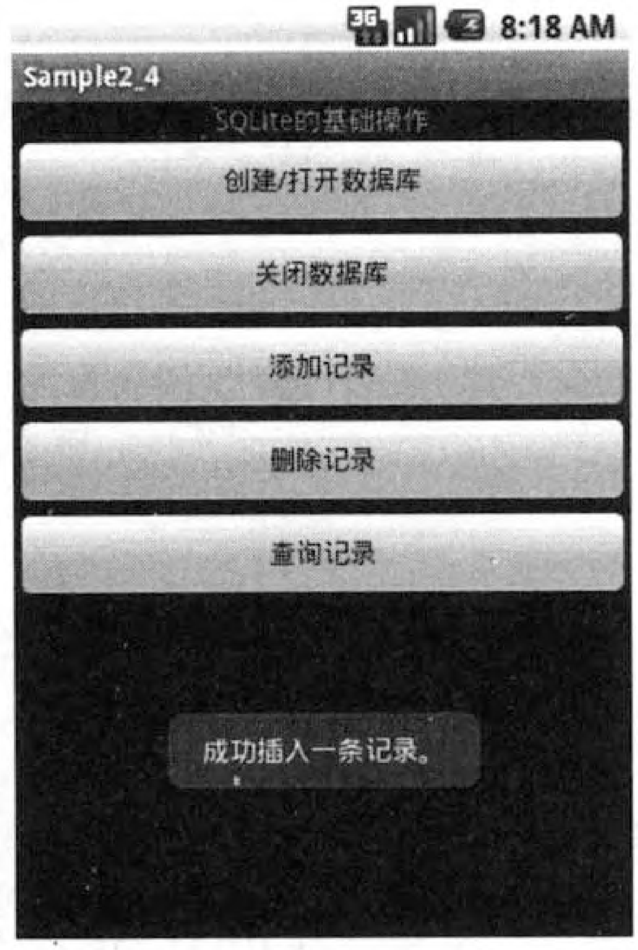

8:18 AM
Sample2_4
SQLite的基础操作
创建/打开数据库
关闭数据库
添加记录
删除记录
查询记录
成功创建数据库。

图2-10　创建数据库


8:18 AM
Sample2_4
SQLite的基础操作
创建/打开数据库
关闭数据库
添加记录
删除记录
查询记录
成功插入一条记录。
图2-11　插入记录

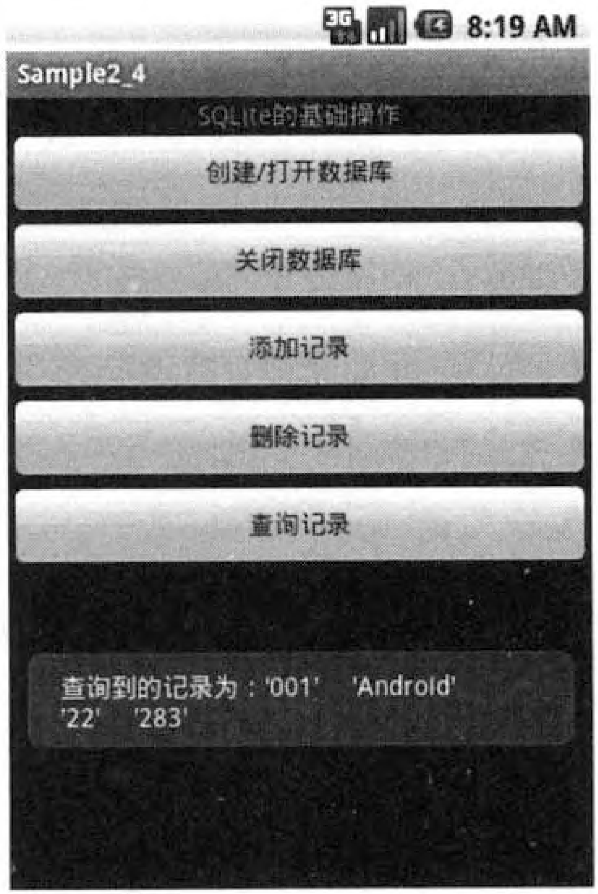


图2-12　查询记录

介绍完本案例的运行效果后，接下来将开发本案例中唯一的一个类——Sample2_4_Activity，其代码如下。

代码位置：见随书光盘中源代码/第2章/Sample2_4/src/com/bn/pp4目录下的Sample2_4_Activity.java。

```
package com.bn.pp4;                                    //声明包
import android.app.Activity;                           //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.widget.Toast;                           //引入相关类
public class Sample2_4_Activity extends Activity {
    SQLiteDatabase sld;                                // 声明SQLiteDatabase引用
    public void onCreate(Bundle savedInstanceState) {  // 重写的onCreate方法
        super.onCreate(savedInstanceState);
```

```java
        setContentView(R.layout.main);                  // 跳转到主界面
            Button b = (Button) this.findViewById(R.id.Button01);
                                                        // 获取打开/创建数据库按钮的引用
            b.setOnClickListener(                       // 为打开/创建按钮添加监听器
            new OnClickListener() {
                public void onClick(View v) {
                    createOrOpenDatabase();             // 调用方法打开或创建数据库
            }});
            b = (Button) this.findViewById(R.id.Button02);  // 获取关闭数据库按钮的引用
            b.setOnClickListener(                       // 为关闭按钮添加监听器
            new OnClickListener() {
                public void onClick(View v) {
                    closeDatabase();                    // 调用方法关闭数据库
            }});
            b = (Button) this.findViewById(R.id.Button03);  // 获取添加记录按钮的引用
            b.setOnClickListener(                       // 为添加按钮添加监听器
            new OnClickListener() {
                public void onClick(View v) {
                    insert();                           // 调用方法插入记录
            }});
            b = (Button) this.findViewById(R.id.Button04);  // 获取删除记录按钮的引用
            b.setOnClickListener(                       // 为删除按钮添加监听器
            new OnClickListener() {
                public void onClick(View v) {
                    delete();                                   // 调用方法删除记录
            }});
            b = (Button) this.findViewById(R.id.Button05);  // 获取查询记录按钮的引用
            b.setOnClickListener(                       // 为查询按钮添加监听器
            new OnClickListener() {
                @Override
                public void onClick(View v) {
                    query();                            // 调用方法查询记录
        }});}
        public void createOrOpenDatabase() {            // 创建或打开数据库的方法
            try {
                sld = SQLiteDatabase.openDatabase(
                        "/data/data/com.bn.pp4/mydb",  // 数据库所在路径
                        null, // 游标工厂，默认为 null
                        SQLiteDatabase.OPEN_READWRITE |
                        SQLiteDatabase.CREATE_IF_NECESSARY //模式为读写，不存在则创建
                );
                                                        // 生成创建数据库的 SQL 语句
                String sql = "create table if not exists student" +
                        "(sno char(5),stuname varchar(20)," +
                        "sage integer,sclass char(5))";
                sld.execSQL(sql);                       // 执行 SQL 语句
                Toast.makeText(getBaseContext(), "成功创建数据库。",
                        Toast.LENGTH_LONG).show();
            } catch (Exception e) {
                e.printStackTrace();
        }}
```

```
59       public void closeDatabase() {                    // 关闭数据库的方法
60           try {
61               sld.close();                             // 关闭数据库
62               Toast.makeText(getBaseContext(), "成功关闭数据库。",
63                       Toast.LENGTH_LONG).show();
64           } catch (Exception e) {
65               e.printStackTrace();
66       }}
67       public void insert() {                           // 插入记录的方法
68           try  {                                       // 生成插入记录的 SQL 语句
69               String sql = "insert into student values" +
70                       "('001','Android',22,'283')";
71               sld.execSQL(sql);                        // 执行 SQL 语句
72               Toast.makeText(getBaseContext(), "成功插入一条记录。",
73                        Toast.LENGTH_LONG).show();
74           } catch (Exception e) {
75               e.printStackTrace();
76       }}
77       public void delete() {                           // 删除记录的方法
78           try  {                                       // 生成删除所有记录的 SQL 语句
79               String sql = "delete from student;";
80               sld.execSQL(sql);                        // 执行 SQL 语句
81               Toast.makeText(getBaseContext(), "成功删除所有记录。",
82                       Toast.LENGTH_LONG).show();
83           } catch (Exception e) {
84               e.printStackTrace();
85       }}
86       public void query(){                             // 查询的方法
87           try {                                        // 生成查询记录的 SQL 语句
88               String sql = "select * from student where sage>?";
89               Cursor cur = sld.rawQuery(sql, new String[] { "20" });
                                                          // 获取 Cursor 对象引用
90               while (cur.moveToNext()) {               // 若存在记录
91                   String sno = cur.getString(0);       // 获取第一列信息
92                   String sname = cur.getString(1);     // 获取第二列信息
93                   int sage = cur.getInt(2);            // 获取第三列信息
94                   String sclass = cur.getString(3);    // 获取第四列信息
95                   Toast.makeText(
96                           getBaseContext(),
97                           "查询到的记录为：'" + sno + "'\t'" + sname
98                           + "'\t\t'" + sage+ "'\t'" + sclass + "'",
99                           Toast.LENGTH_LONG).show();
100              }
101              cur.close();                             // 关闭 Cursor
102          } catch (Exception e) {
103              e.printStackTrace();
104  }}}
```

- 第10-40行为案例中的各个按钮添加监听器，监听器中调用对应的方法来实现数据库的打开/创建、关闭、插入、删除和查询等操作。
- 第41-58行为创建及打开数据库的方法，方法中首先获取了SQLiteDatabase对象的引用，并为其指定数据库的存储路径和读写模式，然后用“create table”语句来创建了一张名称为student的表。
- 第67-76行为向数据库中插入一条记录的方法，插入的记录内容为“‘001’，‘Android’，22，‘283’”。
- 第77-85行为删除数据库中所有记录的方法。

• 第86-104行为从数据库中查找符合条件记录的方法，首先需要获取Cursor对象的引用，并为其添加查找范围（具体范围为年龄大于20）。若查到相应记录，则将记录信息用Toast显示出来。

2.3.4 使用ContentProvider组件共享数据

前一小节介绍了SQLite数据库中的一些操作，但有时数据库中的信息不但创建其的应用程序要使用，还希望能够分享给其他应用程序使用。这时就需要使用ContentProvider组件了，ContentProvider组件的基本情况如下所列。

• Android平台中每个应用程序都有自己的用户ID并在自己的进程中运行，每个进程拥有独立的运行环境，这样可以保证程序的完整性。但这也使得应用程序在需要进行资源共享和数据通信时，很不方便。为了解决这一问题，Android提供了专门用来在应用程序之间分享数据的ContentProvider组件。

• ContentProvider能将应用程序中特定的数据提供给其他应用程序使用，这些数据可以来自应用程序私有文件夹下的私有数据文件，也可以来自应用程序自己私有的SQLite数据库。当然，数据的来源还有很多其他选择，ContentProvider组件本身并没有做出限制，读者可以充分发挥想象的空间。

• 使用ContentProvider组件共享数据的基本方式是继承ContentProvider类并重写其中的相应方法，具体情况在后面的案例中进行介绍。

• 别的应用程序想分享数据时需要使用ContentResolver，通过ContentResolver对象将分享的需求发送给ContentProvider组件，而不能直接调用ContentProvider组件。

下面使用ContentProvider组件将上一小节的案例进行升级，使得此案例具有分享数据给其他应用程序的能力，其具体开发步骤如下。

（1）在案例Sample2_4的com/bn/pp4包下新建MyContentProvider类，该类继承自ContentProvider类，并要实现其中所有的抽象方法，具体代码如下。

代码位置：见随书光盘中源代码/第2章/Sample2_4/src/com/bn/pp4目录下的MyContentProvider.java。

```
package com.bn.pp4;                                          //声明包
import android.content.ContentProvider;                      //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.net.Uri;                                      //引入相关类
public class MyContentProvider extends ContentProvider {     // 继承ContentProvider
    private static final UriMatcher um;                      // 声明Uri匹配引用
    static {
        um = new UriMatcher(UriMatcher.NO_MATCH);            // 创建UriMatcher
```

```
9        um.addURI("com.bn.pp4.provider.student", "stu", 1); // 设置匹配字符串
10       }
11       SQLiteDatabase sld;              // 声明 SQLiteDatabase 引用
12       public String getType(Uri uri) {
13           return null;
14       }
15                                        // 调用数据库的 query 方法时会自动调用该方法
16       public Cursor query(Uri uri, String[] projection, String selection,
17               String[] selectionArgs, String sortOrder) {
18           switch (um.match(uri)) {                 // 若匹配成功
19           case 1:                                  // 执行操作,获取 Cursor 对象引用
20               Cursor cur = sld.query("student", projection, selection,
21                       selectionArgs, null, null, sortOrder);
22               return cur;                          // 返回 Cursor 对象引用
23           }
24           return null;
25       }
26       public int delete(Uri arg0, String arg1, String[] arg2) {    // 空实现
27           return 0;
28       }
29       public Uri insert(Uri uri, ContentValues values) {           // 空实现
30           return null;
31       }
32       public boolean onCreate() {                  // 创建数据库时自动调用该方法
33           sld = SQLiteDatabase.openDatabase(
34                   "/data/data/com.bn.pp4/mydb",    // 数据库所在路径
35                   null,                            // 游标工厂，默认为 null
36                   SQLiteDatabase.OPEN_READWRITE|
37                   SQLiteDatabase.CREATE_IF_NECESSARY // 读写、若不存在则创建
38           );
39           return false;
40       }
41       public int update(Uri uri, ContentValues values, String selection,
42               String[] selectionArgs) {            // 空实现
43           return 0;
44       }
45   }
```

- 第6-10行为声明Uri匹配对象，并且设置匹配字符串。此匹配字符串在需要得到分享数据的应用程序中提供给ContentResolver使用，以进行配对。
- 第12-14行重写了getType方法，本案例中对getType方法没有要求，因此其返回空值。•第16-25行重写了query方法，在匹配成功后，数据需求方通过ContentResolver调用此方法查询需要的数据。
- 第26-31行重写了delete与insert方法，本案例中对这两个方法没有要求，因此都设置为返回空值。

• 第32-40行重写了onCreate方法，其功能为首先获取SQLiteDatabase对象引用，然后创建或打开数据库，为信息的分享做好准备。

• 第41-44行重写了update方法，本案例中对这个方法没有要求，因此设置为返回空值。

（2）仅仅是完成上面的代码还是不够的，在Android程序开发中，有一个很重要的配置文件AndroidManifest.xmlo要想使用ContentProvider组件，在完成代码的开发后，还必须在该配置文件中进行相应的配置，将如下代码插入到AndroidManifest.xml文件中的application标签中。

代码位置：见随书光盘中源代码/第2章/Sample2_4目录下的

AndroidManifest.xml。

```
<provider
    android:name="MyContentProvider"                        <!--将调用的类名-->
    android:authorities="com/bn/pp4.provider.student"    <!--要匹配的Uri字符串-->
/>
```

2.3.5 使用ContentResolver获取分享数据

升级完了Sample2_4使其具有了数据分享能力之后，就可以在别的应用程序中通过ContentResolver匹配到Sample2_4案例中的ContentProvider组件获取分享的数据了。具体的开发步骤如下所列。

（1）新建项目Sample2_4_From，将项目的包名设定为com.bn.pp4f，并新建一个继承自Activity的类ContentConsumerActivity，其代码如下。

代码位置：见随书光盘中源代码/第2

章/Sample2_4_From/src/com/bn/pp4f目录下的ContentConsumerActivity.java。

```
package com.bn.pp4f;                                        //包声明
import android.app.Activity;                                //相关类的引入
//……此处省略了部分相关类的引入代码，读者可自行查看随书光盘的源代码
import android.widget.EditText;                             //相关类的引入
public class ContentConsumerActivity extends Activity {
    ContentResolver cr;                                     // ContentResolver 的引用
    @Override                                               //重写方法的标志
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);                 //继承父类的 onCreate 方法
            setContentView(R.layout.main);                  //跳转到主界面
            cr=this.getContentResolver();           //获取 ContentResolver 的对象
            //初始化查询按钮
            Button b=(Button)this.findViewById(R.id.Button01);// Button 类的引用
            b.setOnClickListener(                           //设置按钮监听
              new OnClickListener(){
                  @Override                                 //重写方法的标志
                  public void onClick(View v) {             //重写 onClick 方法
                      String stuname="Android";             //设置查询的字符串
                      Cursor cur=cr.query(
                        Uri.parse("content://com.bn.pp4.provider.student/stu"),
                        new String[]{"sno","stuname","sage","sclass"},
                        "stuname=?",                        //查询条件
                        new String[]{stuname},
                        "sage ASC"
                      );
                  while(cur.moveToNext()){
                      String sno=cur.getString(0);          //获取学号
```

```
28                  String sname=cur.getString(1);        //获取名称
29                  int sage=cur.getInt(2);               //获取年龄
30                  String sclass=cur.getString(3);       //获取班级
31                  appendMessage(sno+"\t"+sname+"\t\t"+sage+"\t"+sclass);
32              }
33              cur.close();                              //关闭ContentResolver
34      }});}
35      public void appendMessage(String msg){             //向文本区中添加文本
36       EditText et=(EditText)this.findViewById(R.id.EditText02);//获取EditText的对象
37       et.append(msg+"\n");                              //添加显示的字符串
38  }}
```

- 第8-25行主要功能为获取ContentResolver对象的引用，并给按钮添加监听器，使得按钮按下后可以通过ContentResolver匹配到Sample2_4案例中的ContentProvider组件获取需要的数据。
- 第26-33行功能为将获取的Sample2_4案例分享的数据显示到屏幕上的EditText控件中。
- 第35-38行为向EditText控件中添加文本信息的方法。

（2）Sample2_4_From案例开发完成后，运行该案例，其效果如图2-13、图2-14所示。

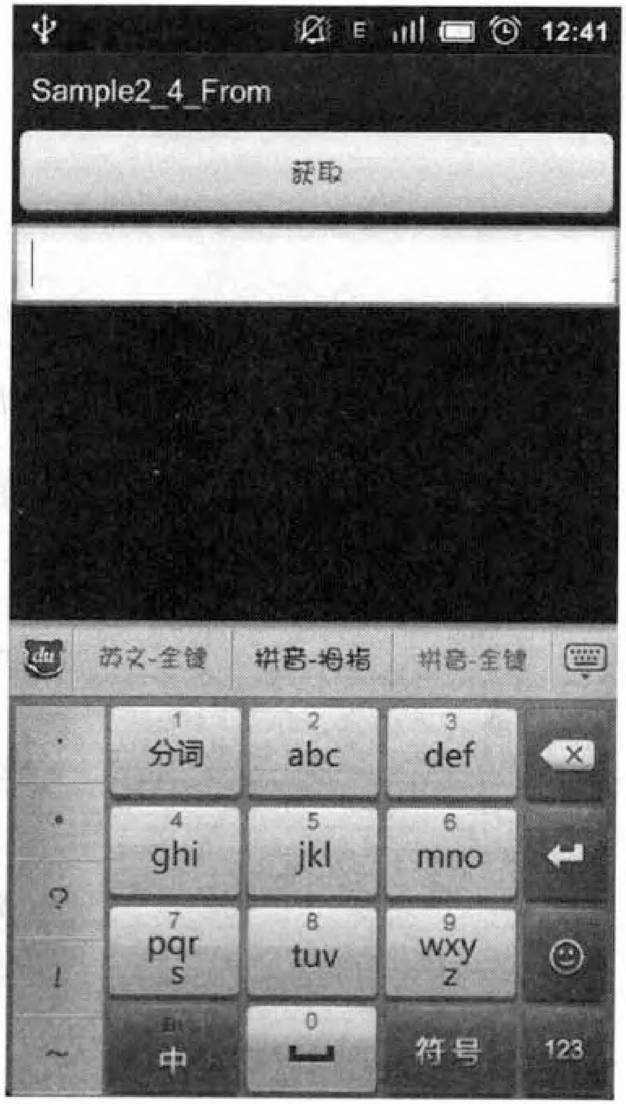

12:41
Sample2_4_From
获取
分词
abc
def
ghi
jkl
mno
pqr
s
tuv
wxy
z
中
符号
123

图2-13　运行界面1



图2-14　运行界面2

> **说明**
>
> 图2-13为运行该案例后的界面效果图，图2-14为单击“获取”按钮后，通过ContentResolver匹配到Sample2_4案例中的ContentProvider组件获取数据后的效果图。

2.4　文件I/O

I/O即输入与输出，几乎所有文件操作的工作都离不开I/O。对于Android游戏开发来说，I/O 操作尤为重要，游戏中的地图数据、人物图片资源等的流畅读取是保证游

戏可玩性的重要方面。因此在开发时，不同的场合必须选用合适的I/O操作方式，才能保证满足用户的需求。

Android中文件I/O分为3种方式：SD卡文件读取、手机中文件夹的访问和应用程序中Assets文件的读取，本节将结合简单的小案例对这几种文件I/O操作进行介绍。

2.4.1 访问SD卡中的文件

伴随着游戏品质的不断提高，游戏数据占用的存储空间也成几何级数增加，以往几十KB、几MB的手机游戏已经发展为几十MB、几百MB甚至几个GB的大型手机游戏。为了适应存储需求的增长，SD卡也一直在进行更新换代，4GB、8GB、16GB大小的SD卡已随处可见。

Android的设计者自然不会忽略这一点，在Android平台上可以轻松地对手机SD卡中的文件进行读取和操作。下面通过一个案例来详细讲解在Android开发中如何访问SD卡中的文件。

1．案例的运行效果

运行本小节案例Sample2_5，在输入框中输入要读取的文件名，然后单击“打开文件”按钮。若SD卡中存在该文本文件，则在文本区域显示出文件的内容，反之则提示未找到该文件。本案例的运行效果如图2-15、图2-16所示。



图2-15　成功加载SD卡中的文件



图2-16　未找到指定的文件

> **提示**
>
> 运行本案例时要注意SD卡中文本文件的编码格式，在Android系统中一般要采用UTF-8编码才能保证没有乱码的出现。编码格式为UTF-8的文本文件见随书光盘中源代码/第2章目录下的AndroidIO.txt，读者可以将该文件导入到手机或模拟器的SD卡中再对本案例进行测试。

2．案例的开发

介绍完本案例的运行效果以及文本文件编码的注意事项后，接下来将对本案例的主控制类 Sample2_5_Activity进行开发，其代码如下。

代码位置：见随书光盘中源代码/第2章/Sample2_5/src/com/bn/pp5目录下的Sample2_5_Activity.java。

```
1package com/bn/pp5;
2import java.io.File;                                    //引入相关包
3public class Sample2_5_Activity extends Activity {      //创建 Activity
4   @Override
5   public void onCreate(Bundle savedInstanceState) {   //重写 onCreate 方法
6       super.onCreate(savedInstanceState);
7       setContentView(R.layout.main);                  //跳转到主界面
8       Button ok=(Button)this.findViewById(R.id.Button01);  //获取打开 Button
9       ok.setOnClickListener(                          //为打开按钮添加监听器
10              new OnClickListener(){
11              public void onClick(View v) {
12                  EditText et1=(EditText)findViewById(R.id.EditText01);
13                  String nr=loadText(et1.getText().toString().trim());
14                  EditText et2=(EditText)findViewById(R.id.EditText02);
15                  et2.setText(nr);                    //设置显示框内容
16          }});}
17      public String loadText(String name){            //加载 SD 卡文件方法
18          String nr=null;                             //内容字符串
19      try{
20          File f=new File("/sdcard/"+name);           //创建对应文件
21          byte[] buff=new byte[(int) f.length()];     //创建响应大小的 byte 数组
22          FileInputStream fis=new FileInputStream(f);
23          fis.read(buff);                             //读入文件
24          fis.close();                                //关闭输入流
25          nr=new String(buff,"utf-8");                //转码生成字符串
26          nr=nr.replaceAll("\\r\\n","\n");            //替换换行符
27      }catch (Exception e) {
28          Toast.makeText(getBaseContext(),            //Toast 提示用户
29                      "对不起，没有找到指定文件。",
30                      Toast.LENGTH_LONG).show();
31      }
32      return nr;                                      //返回内容字符串
33  }}
```

- 第8-16行为打开文件按钮添加监听器，监听器功能为首先获取文本框中用户输入的文件名，然后调用从SD卡加载文本文件的方法，获取SD卡中的对应文件内容并送入EditText控件显示。
- 第17-32行为加载SD卡中文本文件的方法，其入口参数为要加载文件的文件名。工作过程为首先用Java标准的I/O操作来加载指定文本文件中的数据，然后将数据编码为字符串，并替换换行符等空白字符，最后返回加载的内容。

> **提示**
>
> 通过上面的小案例可以看出，Android中对SD卡文件的I/O与标准Java相同，因此开发成本很低，易于上手。

2.4.2 访问手机中的文件夹

前一小节中是将文件存放到SD卡中，其实文件也可以存放在手机内部存储（ROM，相当于PC的硬盘）中。不过在Android系统中，其为每个应用程序在手机的ROM中都分配了一个私有的目录，命名规则为“/data/data/<应用程序的包名>”。例如，应用程序的包名为“com.bn”，则系统分配的私有目录为“/data/data/com.bn”。

应用程序开发时，若需要在ROM中存放文件，一般应该存放到系统分配的私有目录中。启动Eclipse，进入其中的DDMS面板，打开File Explorer，即可查看手机ROM中的文件组织情况，如图2-17所示。

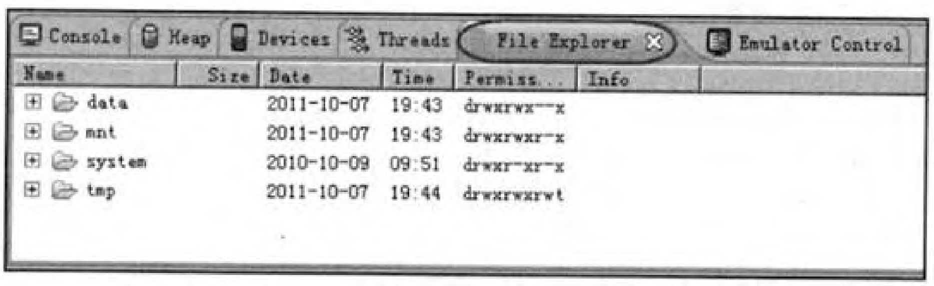


图2-17 查看手机内部存储文件

下面给出一个浏览手机ROM中文件夹的小案例，通过单击文件夹的名称，可以打开相应的文件夹，显示其中的文件列表。运行本案例，其运行效果如图2-18、图2-19和图2-20所示。

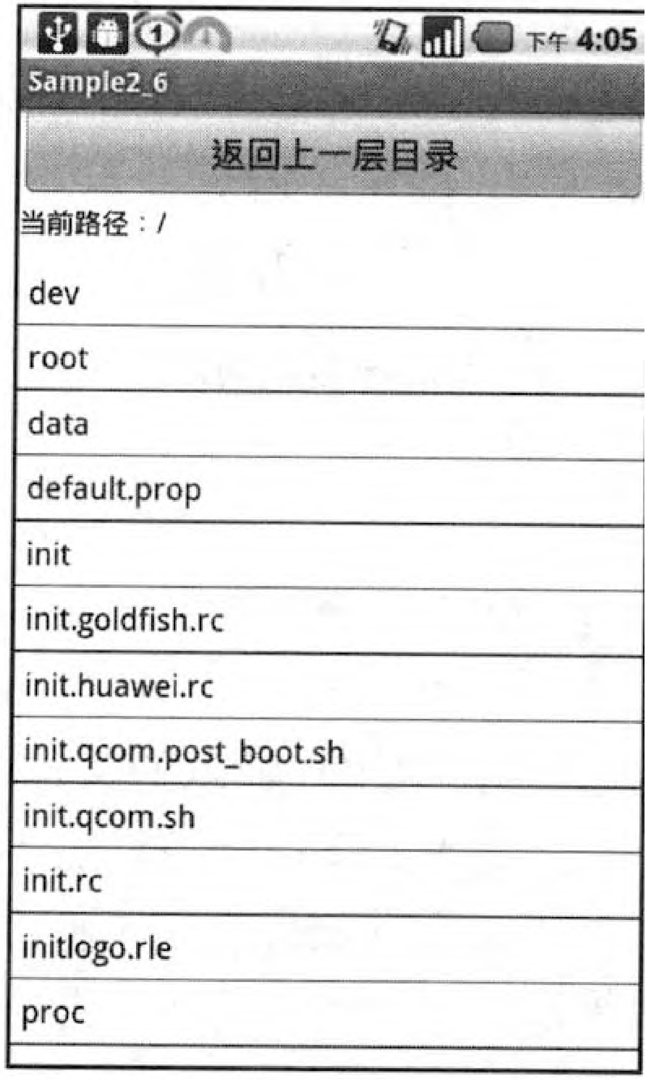

下午 4:05
Sample2_6
返回上一层目录
当前路径：/
dev
root
data
default.prop
init
init.goldfish.rc
init.huawei.rc
init.qcom.post_boot.sh
init.qcom.sh
init.rc
initlogo.rle
proc

图2-18　程序初始运行



图2-19　进入dev目录

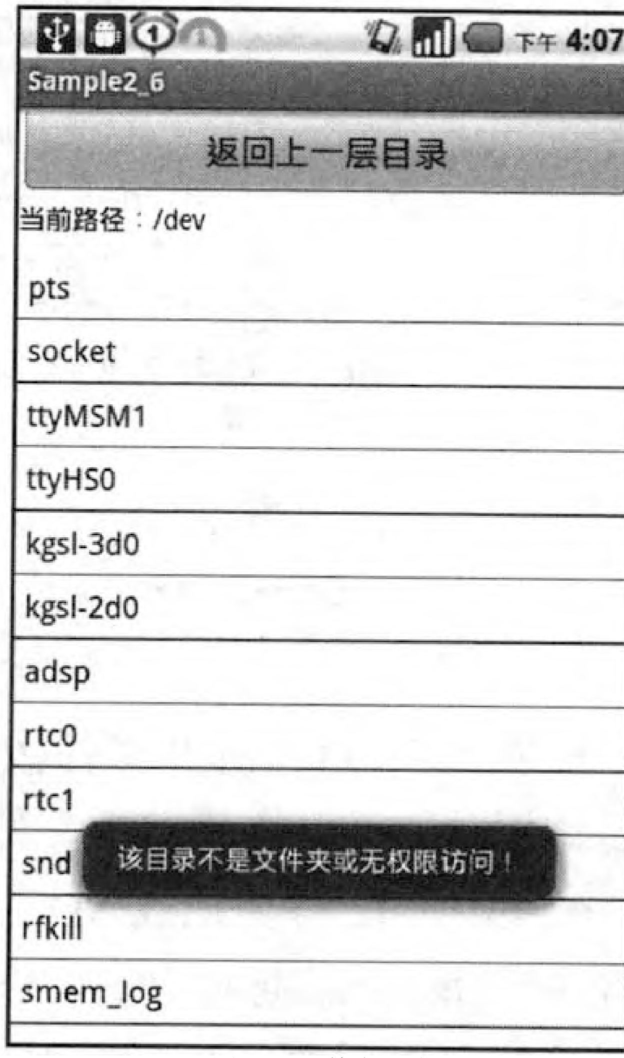


图2-20　单击出错

> **说明**
>
> 图2-18为程序一开始运行，显示“/”目录下内容的情况。图2-19为单击进入了dev目录，图2-20为单击了不允许进入或不是文件夹的条目。

了解了本案例的运行效果之后，接下来对本案例中唯一的类——Sample2_6_Activity进行开发， 具体代码如下。

代码位置：见随书光盘中源代码/第2章/Sample2_6/src/com/bn/pp6目录下的Sample2_6_Activity.java。

```
1 package com.bn.pp6;                                        //声明包
2 import java.io.File;                                       //引入相关类
3 ……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
4 import android.widget.AdapterView.OnItemClickListener;     //引入相关类
5 public class Sample2_6_Activity extends Activity {         // 创建Activity
6     String currPath;                                       // 当前路径字符串
7     String rootPath = "/";                                 // 根目录路径
8     TextView currDirTV;                                    //显示当前路径的TextView引用
9     @Override
10        public void onCreate(Bundle savedInstanceState) {
11            super.onCreate(savedInstanceState);
12            setContentView(R.layout.main);                          // 跳转到主界面
13            final ListView lv = (ListView) this.findViewById(R.id.lv); // 获取ListView
14            Button back = (Button) this.findViewById(R.id.back);     // 获取返回按钮
15            final File[] files = getFiles(rootPath);
                                              // 调用getFiles方法获取根目录下文件列表
16            currDirTV = (TextView) this.findViewById(R.id.currDirTV); // 获取ListView
17            currPath = rootPath;
18            currDirTV.setText("当前路径：" + currPath);        //设置当前路径
19            initListView(files, lv);                          // 初始化显示列表
20            back.setOnClickListener                           // 返回按钮监听器
21            (new OnClickListener() {
22                @Override
23                public void onClick(View v) {
24                    if (!currPath.equals(rootPath)) {// 若当前路径不是根目录，返回到上一层目录
25                        File f = new File(currPath);   // 获取当前路径下的文件列表
26                        f = f.getParentFile();         // 获取当前路径的上层路径
27                        currPath = f.getPath();        // 更改当前路径
28                        currDirTV.setText("当前路径：" + currPath);//设置当前路径
29                        initListView(getFiles(currPath), lv);  // 初始化显示列表
30        }}}});}
31                                                   // 获取当前目录下的文件列表的方法
32        public File[] getFiles(String filePath) {
33            File[] files = new File(filePath).listFiles();  // 获取当前目录下的文件列表
34            return files;                                   // 返回文件列表
35        }
36        ……//此处省略了初始化ListView的方法，读者可以自行查阅随书光盘中的源代码
37    }
```

- 第9-30行为案例整体功能的实现代码。程序运行时，首先调用initListView方法初始化文件显示列表，显示根目录下的文件情况。然后给“返回上一层”按钮添加了监听器，监听器功能为若存在上一层目录，则刷新文件显示列表为上一层目录下的情况。
- 第32-35行为获取当前路径下所有文件列表的方法。该方法中首先需要获得当前路径对应的File对象，之后返回其下对应的文件列表。

提示

由于篇幅所限，初始化文件列表的代码省略，有需要的读者请自行查阅光盘中的源代码进行学习。

2.4.3 读取assets文件夹下的内容

Android还支持将应用程序所需数据文件打包到apk文件中，省去了使用者安装应用程序后还需要下载数据包的情况。但是要注意的是，打包到apk文件中的数据文件并不是可以放在任何位置，一般应该放到项目中的assets文件夹下。

提示

对于特别大（例如50MB）的数据文件，在实际开发中一般还是与apk安装文件分开的。对于不大的数据文件，就非常适合打包到apk包中的assets文件夹下。

下面将通过一个小案例来说明如何编程访问assets文件夹下的内容，其主要功能是完成对应用程序内部的assets文件夹下文件的读取。操作过程为在案例初始界面可以输入文件名，然后单击打开按钮。若存在该文件，则在下方显示出文件内容；反之则弹出Toast提示用户重新输入。案例运行效果如图2-21、图2-22所示。



图2-21　程序运行效果图



图2-22　无法找到指定文件

了解了本案例的功能后，下面来介绍本案例的开发过程，具体步骤如下。

（1）首先将AndroidIO.txt文本文件复制到项目中的assets文件夹中，如图2-23所示。



图2-23　添加文本文件

（2）将AndroidIO.txt文本文件复制到项目中的assets文件夹后，接下来将开发本案例中唯一的类Sample2_7_Activity，其代码如下。

代码位置：见随书光盘中源代码/第2章/Sample2_7/src/com/bn/pp7目录下的Sample2_7_Activity.java。

```
1  package com/bn/pp7;
2  import java.io.ByteArrayOutputStream;                    //引入相关包
3  public class Sample2_7_Activity extends Activity {      //创建Activity
4   public void onCreate(Bundle savedInstanceState) {      //重写onCreate方法
5      super.onCreate(savedInstanceState);
6      setContentView(R.layout.main);                      //跳转到主界面
7      Button ok=(Button)this.findViewById(R.id.Button01); //获取打开按钮的引用
8      ok.setOnClickListener(                              //为打开按钮添加监听器
9        new OnClickListener(){
10             public void onClick(View v){
11                 EditText et1=(EditText)findViewById(R.id.EditText01);
12                 String nr=loadText(et1.getText().toString().trim());
13                 EditText et2=(EditText)findViewById(R.id.EditText02);
14                 et2.setText(nr);//设置显示框内容
15             }});}
16      public String loadText(String name){               //加载assets文件方法
17       String nr=null;                                   //内容字符串
18           try{
19               InputStream is=this.getResources().getAssets().open(name);
20               int ch=0;
21               ByteArrayOutputStream baos=new ByteArrayOutputStream();
22               while((ch=is.read())!=-1) {
23                   baos.write(ch);//读取文件
24               }
25               byte[] buff=baos.toByteArray();           //转化为字节数组
26               baos.close();                             //关闭输入输出流
27               is.close();                               //关闭输入输出流
28               nr=new String(buff,"utf-8");              //转码生成新字符串
29               nr=nr.replaceAll("\\r\\n","\n");          //替换换行符等空白字符
30           }catch (Exception e){                         //没有找到对应文件，进行提示
31               Toast.makeText(getBaseContext(),
32                          "对不起，没有找到指定文件。",
33                          Toast.LENGTH_LONG).show();
34           }
35           return nr;                                    //返回内容字符串
36      }}
```

- 第7-15行首先获取了打开按钮的引用，然后给此按钮添加了监听器。监听器的功能为读取用户输入名称的文件内容，并显示到界面上。

• 第16-36行为从assets中加载指定名称文本文件的方法，其首先获取了指向指定文件的输入流，然后从输入流中读取数据，接着将数据编码为字符串，并替换换行符等空白字符，最后返回加载的内容。

> **提示**
>
> 通过上述几个小节案例的介绍，相信读者对Android平台下文件的存储和读取已经有所掌握。在真正的开发过程中，开发人员应该考虑各方面的因素，在需要时采用最适合自己的存储方式进行存储。

2.5　2D动画的开发

虽然本书是着重介绍3D的开发技术，但在大部分的3D应用中也需要有不少的2D界面，如菜单、帮助等。本节将介绍一般用于开发游戏中2D界面的SurfaceView类的使用。其继承自View类，但与View的不同之处在于，View更新画面必须是在UI线程中（也可以理解为主线程中），而SurfaceView更新画面可以在自定义线程中进行，大大方便了开发。

> **提示**
>
> 关于Android下的多线程问题，读者可以参考笔者在人民邮电出版社出版的《Android应用开发完全自学手册——核心技术、传感器、2D/3D、多媒体与典型案例》一书中第8章的相关内容，那里有比较详细的介绍。

2.5.1　SurfaceView用法简介

实际开发中，一般采用继承SurfaceView进行自定义的方法来开发2D动画效果。开发时不但要继承SurfaceView类，一般还要实现SurfaceHolder.Callback接口。SurfaceView类中绘制界面的方法为onDraw，其具体签名为“protected void onDraw（Canvas canvas）”，每调用一次该方法，就会重新绘制一帧画面。

SurfaceHolder.Callback接口中主要包含了2D界面的3个生命周期相关的回调方法，其方法签名和说明如表2-3所列。

表2-3　SurfaceHolder.Callback接口中的3个生命周期回调方法

方法签名	说　　明
public void surfaceCreated(SurfaceHolder holder)	该方法在SurfaceView被创建时被调用

public void surfaceChanged(SurfaceHolderholder, int format, int width, int height)	该方法在SurfaceView变化时被调用（如自动横竖屏切换时），在surfaceCreated方法被调用后，该方法至少会被调用一次
public void surfaceDestroyed(SurfaceHolder holder)	该方法在SurfaceView销毁时被调用

以上3个生命周期回调方法都有其各自的用途，具体情况如下所列。

- 每次创建界面时需要初始化图片、线程等资源，这些代码一般写在surfaceCreated方法中。
- 当SurfaceView变化时，如果需要改变一些值，这些代码应该放在surfaceChanged方法中。
- SurfaceView被销毁时，有些与界面相关的资源应该被释放掉，这些代码应写在surfaceDestroyed方法中。

2.5.2 使用SurfaceView实现2D动画

上一小节介绍了SurfaceView的基本用法，本小节将通过一个2D动画的简单案例来具体说明SurfaceView在开发中的应用。本案例中的动画实现了这样的场景：一枚炮弹从屏幕的左下角发射，以抛物线的轨迹划过天空，并在一定的位置爆炸，其效果分别如图3-24、图3-25和图3-26所示。



图3-24　炮弹开始飞出

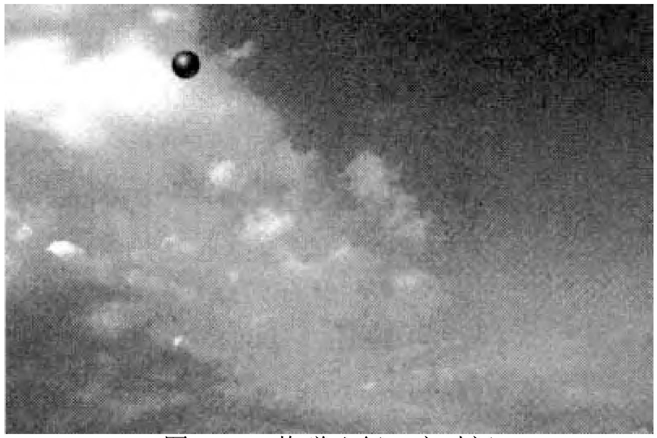


图3-25　炮弹飞行一定时间



图3-26　爆炸效果图

说明

图3-24、图3-25和图3-26从左到右分别为炮弹刚刚发射、炮弹将近飞到最高点和炮弹爆炸时的效果图。

介绍完本案例的运行效果之后，下面将详细讲解案例中各个类代码的具体实现，具体步骤如下。

（1）首先将开发本案例的主控制类——Sample2_8_Activity，该类在程序开始时执行。其主要功能为设置应用程序为全屏及横屏模式，并跳转到动画呈现对应的SurfaceView，具体代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_8/src/com/bn/pp8目录下的Sample2_8_Activity.java。

```
1  package com/bn/pp8;                                          //声明包
2  import android.app.Activity;                                 //引入相关类
3  ……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
4  import android.view.WindowManager;                           //引入相关类
5  public class Sample2_8_Activity extends Activity {
6    MySurfaceView gameView;                                    //游戏界面
7   @Override
8   public void onCreate(Bundle savedInstanceState) {           //重写 onCreate 方法
9      super.onCreate(savedInstanceState);
10           requestWindowFeature(Window.FEATURE_NO_TITLE);     //设置为全屏模式
11           getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN ,
12                    WindowManager.LayoutParams.FLAG_FULLSCREEN);
13                                                              //设置为横屏模式
14         this.setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
15         gameView = new MySurfaceView(this);                  //创建 2D 动画界面对象
16         this.setContentView(gameView);                       //跳转到 2D 动画界面
17   }}
```

- 第10-14行将Activity设置为全屏，并设置为横屏模式。
- 第15-16行创建了用于实现2D动画绘制呈现的MySurfaceView类的对象，并跳转到2D动画界面。这样，程序一启动成功用户看到的将是2D动画界面了。

（2）开发完本案例的主控制类后，接下来将对本案例中用于实现2D动画绘制呈现的MySurfaceView类进行开发。该类继承自SurfaceView，同时实现了SurfaceHolder.Callback接口，具体代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_8/src/com/bn/pp8目录下的MySurfaceView.java。

```
package com/bn/pp8;                                             //声明包
```

```java
import android.graphics.Bitmap;                                  //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.view.SurfaceView;                                 //引入相关类
public class MySurfaceView extends SurfaceView implements SurfaceHolder.Callback {
    Sample2_8_Activity activity;                                 //activity 的引用
    Paint paint;                                                 //画笔引用
    DrawThread drawThread;                                       //绘制线程引用
    Bitmap bgBmp;                                                //背景图片
        Bitmap bulletBmp;                                        //炮弹位图
        Bitmap[] explodeBmps;                                    //爆炸位图数组
        Bullet bullet;                                           //炮弹对象引用
        public MySurfaceView(Sample2_8_Activity activity) {     //构造器
            super(activity);
            this.activity = activity;
            this.requestFocus();                                 //获取焦点
            this.setFocusableInTouchMode(true);                  //设置为可触控
            getHolder().addCallback(this);                       //注册回调接口
        }
        @Override
        protected void onDraw(Canvas canvas) {                   //绘制界面的方法
            super.onDraw(canvas);
            canvas.drawBitmap(bgBmp, 0, 0, paint);               //绘制背景
            bullet.drawSelf(canvas, paint);                      //绘制炮弹
        }
        @Override
        public void surfaceChanged(SurfaceHolder holder,int format,int width,int
        height){ }
        @Override
        public void surfaceCreated(SurfaceHolder holder){
            paint = new Paint();                                 // 创建画笔
            paint.setAntiAlias(true);                            // 打开抗锯齿
                                                                 //加载图片资源
           bulletBmp = BitmapFactory.decodeResource(this.getResources(), R.drawable.
            bullet);
            bgBmp = BitmapFactory.decodeResource(this.getResources(), R.drawable.bg);
            explodeBmps=new Bitmap[]{
                    BitmapFactory.decodeResource(this.getResources(), R.drawable.
                    explode0),
                    BitmapFactory.decodeResource(this.getResources(), R.drawable.
                    explode1),
                    BitmapFactory.decodeResource(this.getResources(), R.drawable.
                    explode2),
                    BitmapFactory.decodeResource(this.getResources(), R.drawable.
                    explode3),
                    BitmapFactory.decodeResource(this.getResources(), R.drawable.
                    explode4),
                    BitmapFactory.decodeResource(this.getResources(), R.drawable.
                    explode5),
            };
            bullet = new Bullet(this, bulletBmp,explodeBmps,0,290,1.3f,-5.9f);
                                                                 //创建炮弹对象
```

```
44            drawThread = new DrawThread(this);              //创建绘制线程
45            drawThread.start();                             //启动绘制线程
46        }
47        @Override
48        public void surfaceDestroyed(SurfaceHolder holder) {  //界面销毁时调用的方法
49            drawThread.setFlag(false);                      //停止绘制线程
50    }}
```

- 第6-12行声明了画笔、绘制线程、位图资源、炮弹等对象的引用。
- 第13-19行为MySurfaceView类的构造器，其主要功能为获得焦点、设置为可触控，并注册了生命周期回调接口。
- 第21-25行为重写的onDraw方法，该方法主要功能为呈现2D动画的每一帧。
- 第27行实现了surfaceChanged方法。由于本案例中当SurfaceView变化时，没有需要做的工作，因此该方法为空实现。
- 第29-46行实现了surfaceCreated方法。在该方法中创建或加载了必要的资源，如创建画笔、加载图片、创建炮弹类、创建并绘制启动线程等。
- 第48-50行为surfaceDestroyed方法的实现。由于SurfaceView被销毁时，绘制线程还未关闭，因此应在该方法中停止绘制线程。

（3）开发完本案例的主控制类以及2D动画呈现类之后，接下来将开发绘制线程类——DrawThread。该类主要功能为每隔一定的时间调用onDraw方法，绘制动画中的每一帧，具体代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_8/src/com/bn/pp8目录下的DrawThread.java。

```java
package com/bn/pp8;                                        //声明包
import android.graphics.Canvas;                            //引入相关类
import android.view.SurfaceHolder;                         //引入相关类
public class DrawThread extends Thread {
    private boolean flag = true;                           //线程工作标志位
    private int sleepSpan = 100;                           //线程休眠时间
    MySurfaceView gameView;                                //父界面引用
    SurfaceHolder surfaceHolder;                           //surfaceHolder 引用
    public DrawThread(MySurfaceView gameView) {            //构造器
            this.gameView = gameView;
            this.surfaceHolder = gameView.getHolder();     //创建 SurfaceHolder 对象
        }
        public void run(){
            Canvas c;                                      //声明画布
            while (this.flag){                             //循环执行刷帧任务
                c = null;
                try {
                    c = this.surfaceHolder.lockCanvas(null);//锁定画布
                    synchronized (this.surfaceHolder) {    //锁定 surfaceHolder
                        gameView.onDraw(c);                //绘制一帧画面
                } }finally{
                    if (c != null){                        //释放锁
                        this.surfaceHolder.unlockCanvasAndPost(c);
                } }
                try{
                    Thread.sleep(sleepSpan);               //线程睡眠指定毫秒数
                }catch (Exception e){
                    e.printStackTrace();                   //打印错误堆栈信息
        }}}
        public void setFlag(boolean flag) {                //设置工作标志位的方法
            this.flag = flag;
    }}
```

- 第5-8行为该线程类的成员变量，其中flag为线程是否继续工作的标志，sleepSpan为每两次绘制的时间间隔。
- 第9-12行为该类的构造器，在构造器中获得了gameView及其surfaceHolder对象。
- 第13-29行为实现该线程任务的run方法，其中用一个while循环不断调用gameView的onDraw方法进行刷帧。
- 第30-32行为设置线程工作标志位的setFlag方法。调用该方法并传递参数false，可以停止该线程的绘制工作。

（4）接下来将要开发的是炮弹类——Bullet，此类每个对象表示一枚炮弹，具体代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_8/src/com/bn/pp8目录下的Bullet.java。

```
package com/bn/pp8;                                           //声明包
import android.graphics.Bitmap;                               //引入相关类
import android.graphics.Canvas;                               //引入相关类
import android.graphics.Paint;                                //引入相关类
public class Bullet {
    MySurfaceView gameView;
    private Bitmap bitmap;                                    // 位图
    private Bitmap[] bitmaps;                                 // 爆炸动画图组
    float x;                                                  // x轴位置
        float y;                                              // y轴位置
        float vx;                                             // x轴速度
        float vy;                                             // y轴速度
        private float t = 0;                                  // 生存时间
        private float timeSpan = 0.5f;                        // 时间间隔
        int size;                                             // 炮弹尺寸
        boolean explodeFlag = false;                          // 是否绘制炮弹的标记
        Explosion mExplosion;                                 // 爆炸对象引用
        public Bullet(MySurfaceView gameView, Bitmap bitmap, Bitmap[] bitmaps,
                float x, float y, float vx, float vy) {
            this.gameView = gameView;
            this.bitmap = bitmap;                             // 初始化炮弹的图片
            this.bitmaps = bitmaps;                           // 初始化爆炸动画图片数组
            this.x = x;                                       // 初始化炮弹的位置
            this.y = y;
            this.vx = vx;                                     // 初始化炮弹的速度
            this.vy = vy;
```

```
27            size = bitmap.getHeight();                        // 获得图片的高度
28        }
29        public void drawSelf(Canvas canvas, Paint paint) {  // 绘制炮弹的方法
30            if (explodeFlag && mExplosion != null) {    // 如果已经爆炸，绘制爆炸动画
31                mExplosion.drawSelf(canvas, paint);
32            } else {
33                go();                                         // 炮弹前进
34                canvas.drawBitmap(bitmap, x, y, paint);     // 绘制炮弹
35        } }
36        public void go() {                                    // 炮弹前进的方法
37            x += vx * t;                                      // 水平方向匀速直线运动
38            y += vy * t + 0.5f * Constant.G * t * t;         // 竖直方向上抛运动
39            if (x >= Constant.EXPLOSION_X || y >= Constant.SCREEN_HEIGHT) {//特定位置爆炸
40                mExplosion = new Explosion(gameView, bitmaps, x, y);// 创建爆炸对象
41                explodeFlag = true;                           // 不再绘制炮弹
42                return;
43            }
44            t += timeSpan;                                    // 更新生存时间
45    }}
```

- 第18-28行为该类的构造器，在构造器中进行成员变量的初始化工作。
- 第29-35行为绘制炮弹的方法。如果炮弹已经爆炸，则绘制爆炸动画；否则先调用go方法改变炮弹的位置，再根据位置参数的值绘制炮弹。
- 第36-45行定义了进行炮弹运动计算的go方法。每调用一次该方法，就根据物理公式更新一次炮弹的位置。炮弹位置的计算分为水平（x 轴）和垂直（y 轴）两个方向进行，水平方向为匀速直线运动，垂直方向为上抛运动。当炮弹运动到特定位置时，创建爆炸对象，并标记不再绘制炮弹。

（5）接着对本案例中的Explosion类进行开发，该类主要在炮弹爆炸时使用，实现炮弹爆炸动画中每一帧图片的切换，具体代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_8/src/com/bn/pp8目录下的Explosion.java。

```
1 package com/bn/pp8;                                   //声明包
2 import android.graphics.Bitmap;                       //引入相关类
3 import android.graphics.Canvas;                       //引入相关类
4 import android.graphics.Paint;                        //引入相关类
5 public class Explosion {
6     MySurfaceView gameView;
7     private Bitmap[] bitmaps;                         // 位图
8     float x;                                          // x轴位置
9     float y;                                          // y轴位置
10        private int anmiIndex = 0;                    // 爆炸动画帧索引
11        public Explosion(MySurfaceView gameView, Bitmap[] bitmaps, float x, float y) {
12            this.gameView = gameView;                 //初始化MySurfaceView对象
13            this.bitmaps = bitmaps;
14            this.x = x;                               //初始化x位置
15            this.y = y;                               //初始化y位置
16        }
17        public void drawSelf(Canvas canvas, Paint paint) {      // 绘制背景的方法
18            if (anmiIndex >= bitmaps.length - 1) { // 如果动画播放完毕，不再绘制爆炸效果
19                return;
20            }
21            canvas.drawBitmap(bitmaps[anmiIndex], x, y, paint); // 绘制数组中某一幅图
22            anmiIndex++;                          // 当前下标加1
23 }}
```

- 第11-16行为该类的构造器，在构造器中进行了各成员变量的初始化工作。
- 第17-23行为绘制爆炸动画的方法。根据anmilndex的值，选择图片数组中的某一幅图片进行绘制，然后anmilndex自加1。

> **提示**
>
> 从上面的代码中读者可以看出，其实爆炸动画是由一帧一帧单独的画面组成的。绘制线程定时调用绘制方法绘制动画中的每一帧，动画就呈现在用户眼前。

（6）最后需要开发的是本案例中存放一些常量的类——Constant，其代码如下。

```java
package com.bn.pp8;
public class Constant {                                  //用于统一管理常量的类
    public static final int SCREEN_WIDTH=480;   //屏幕宽度
    public static final int SCREEN_HEIGHT=320;  //屏幕高度
    public static final int EXPLOSION_X=270;    //爆炸x位置
    public static final float G = 1.0f;         //重力加速度的值
}
```

提示

从上面的代码中可以看出，Constant中包含了一些程序运行过程中需要的常量。在实际项目的开发中，将常量独立到一个类中进行管理是非常好的习惯，这很有利于提高代码的可维护性，降低维护的成本。

2.6　Socket网络通信

伴随着移动互联网的高速增长，智能手机上的游戏娱乐应用也逐步由单机向网络化方向发展。因此对于Android开发人员来说，开发具有网络功能的应用程序成为了一项必备的技能。本节将向读者介绍网络开发中非常重要的一项技术，Socket（网络套接字）应用开发。

2.6.1　Socket开发基本知识

网络应用程序的开发也有很多不同的架构与模式，常见的有B/S、C/S等，Socket主要是用于进行传统的C/S模式应用的开发。C/S模式中主要需要开发两个端，服务端与客户端，其基本的工作原理如图2-27所示。

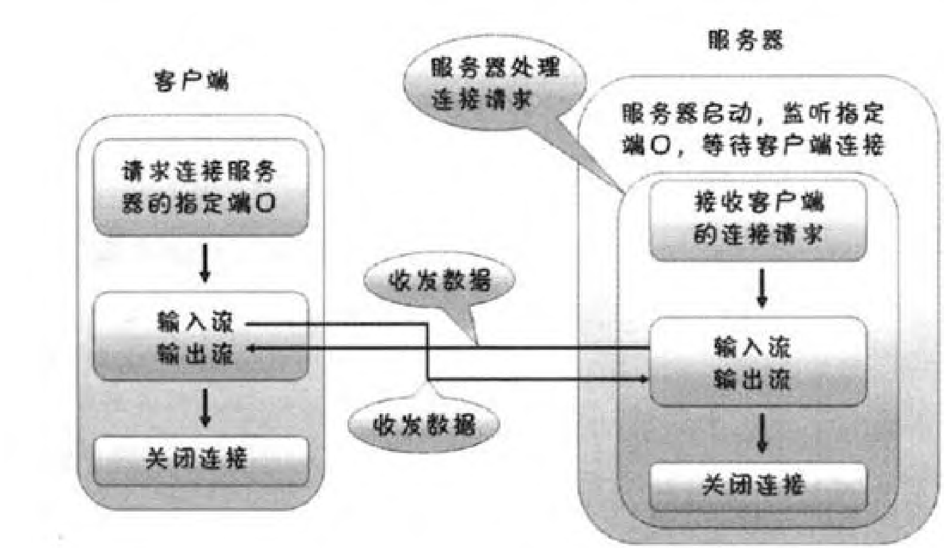


图2-27　C/S模式通信的基本原理

从图2-27中可以看出，C/S模式通信的基本过程如下。

（1）首先启动服务器，监听指定端口，等待接收客户端的连接请求。

（2）客户端请求连接到服务器的指定端口。

（3）服务器收到客户端的连接请求，建立连接。

（4）客户端和服务器同时各打开一个输入流和输出流，客户端的输出流与服务器的输入流连接，服务器的输出流与客户端的输入流连接。

（5）客户端与服务器端通过输入输出流进行双向的消息通信。

（6）当通信完毕后，客户端和服务器同时关闭本次连接。

由于Android应用程序是使用Java进行开发，因此在Android平台下开发基于Socket的C/S模式程序非常简单，直接使用Java中的Socket与ServerSocket类即可。这与传统Java的网络套接字开发几乎没有区别，对于熟悉Java的大部分读者来说非常简单。

2.6.2　服务器端

上一小节简单介绍了基于Socket的C/S模式网络程序的基本工作原理，从本小节开始将向读者介绍一个基于Socket的简单案例。首先介绍服务端的开发，由于本案例比较简单，仅有一个类——Sample2_9_Server，其代码如下。

代码位置：见随书光盘中源代码/第2章/Sample2_9_Server/src/com/bn/Sample2_9_Server目录下的Sample2_9_Server.java。

```
1 package com.bn.Sample2_9_Server;                          //声明包名
2 import java.io.DataInputStream;                          //导入相关类
3……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
4 import java.net.Socket;                                  //导入相关类
5 public class Sample2_9_Server{
6     static ServerSocket sSocket;                         // ServerSocket的引用
7     public static void main(String[] args){              //主方法
8         try{
9             sSocket=new ServerSocket(8877);              //创建ServerSocket对象
10                System.out.println("监听8877接口……");     //打印信息
11                while(true){                  //服务器启动后一直循环服务于不同的客户端
12                    Socket socket=sSocket.accept();        //创建Socket对象
13                    DataInputStream diStream=new DataInputStream(socket.
                      getInputStream());
14                DataOutputStream dotStream=new DataOutputStream(socket.
                  getOutputStream());
15                    System.out.println("客户端信息："+diStream.readUTF());
16                    dotStream.writeUTF("成功连接服务器端");   //写入到输出流中
17                    diStream.close();                     //关闭输入流
18                    dotStream.close();                    //关闭输出流
19                    socket.close();                       //关闭Socket套接字
20            } }catch(Exception e){                        //捕获并打印异常信息
21                e.printStackTrace();
22    } } }
```

- 第9行创建了ServerSocket对象，监听在服务器的8877端口。
- 第11-20行为服务循环，服务器一旦成功启动监听在指定的端口就进入此循环，按照指定的流程服务于一个一个的客户端。本案例中的服务流程为：首先与客户端建立双向数据流，然后首先接收客户端的信息，再向客户端返回信息。

运行本案例，其运行效果如图2-28所示。

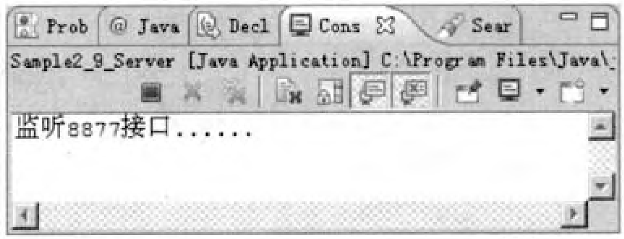


图2-28　对8877接口进行监听

2.6.3　客户端

开发完本案例的服务器端后，接下来在本小节中将主要介绍客户端的开发。由于本案例功能简单，因此客户端也仅有一个类——Sample2_9_ClientActivity，其代码如下。

代码位置：见随书光盘中源代码/第2章/Sample2_9_Client/src/com/bn/Sample2_9_Client目录下的Sample2_9_ClientActivity.java。

```
package com.bn.Sample2_9_Client;                    //声明包名
import java.io.DataInputStream;                     //导入相关类
......//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.widget.TextView;                     //导入相关类
public class Sample2_9_ClientActivity extends Activity{ //创建继承Activity的主控制类
  @Override
  public void onCreate(Bundle savedInstanceState){      //重写的onCreate方法
     super.onCreate(savedInstanceState);
     setContentView(R.layout.main);                     //跳转到主界面
         Button button=(Button)findViewById(R.id.button);  //获得Button按钮的对象
         button.setOnClickListener(                     //为Button按钮添加监听器
           new OnClickListener(){
                  public void onClick(View v){          //重写的onClick方法
                      connectServer();                  //调用connectServer连接服务器
        } } ); }
      public void connectServer(){                      //自定义的连接服务器的方法
           String serverIp="192.168.0.103";             //声明服务器端IP
           try{
                Socket socket=new Socket(serverIp,8877); //创建Socket套接字，发出连接请求
                DataInputStream din=new DataInputStream(socket.getInputStream());
                DataOutputStream dout=new DataOutputStream(socket.getOutputStream());
```

```
22            EditText et=(EditText)this.findViewById(R.id.et);
                                              //获得 EditText 输入对话框对象
23            String tempStr=et.getText().toString();    //获取该对话框中的信息
24            dout.writeUTF(tempStr);          //将信息写入到输出流中
25            TextView tv=(TextView)this.findViewById(R.id.tv); //获得 TextView 的对象
26            tv.setText(din.readUTF());        //将输入流中的数据在 TextView 中显示
27            din.close();                      //关闭输入流
28            dout.close();                     //关闭输出流
29            socket.close();                   //关闭 Socket 套接字
30        }catch(Exception e){                  //捕获并打印异常信息
31            e.printStackTrace();
32    }}}
```

• 第7-15行为继承Activity后重写的onCreate方法，该方法在程序开始时执行。在该方法中首先获得“连接服务器端”按钮对象的引用，之后为该按钮添加监听器，监听器功能为connectServer方法连接服务器收发信息。

• 第16-32行为连接服务器收发信息的connectServer方法，其基本的工作流程为：首先向服务器发出连接请求，连接成功后建立双向数据流，然后首先将用户输入的信息发送给服务器端再从服务器端接收反馈信息。

> **提示**
>
> 在创建Socket套接字对象时需要指明端口号以及服务器端的IP，这里的IP必须为真实IP，不能使用环回地址“127.0.0.1”。因此运行本案例时读者有可能需要根据自己计算机的具体情况修改服务器的IP地址，使得程序可以正常运行。同时，在开发客户端时需要在AndroidManifest.xml中声明联网权限，具体情况参见光盘中的源代码。

运行本案例，在上面的文本框中输入需要发送的信息，然后单击“连接服务器端”按钮。若网络没有问题，则在服务器端可以收到客户端发送的信息，如图2-29所示。同时，客户端也会收到服务器的反馈信息，如图2-30所示。

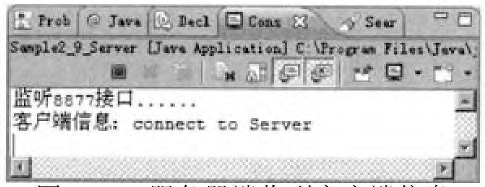


图2-29　服务器端收到客户端信息



图2-30　客户端成功连接到服务器

2.7　蓝牙通信

随着硬件设备价格的不断降低，大部分智能手机上都已配备了蓝牙网络模块，Android设备也是如此。如果能为一些小型的休闲娱乐游戏增加蓝牙联网对战的功能，将会大大增加游戏的可玩性。 本节将向读者详细介绍如何在Android平台下开发具有蓝牙互联功能的应用程序。

2.7.1　蓝牙通信的基本知识

蓝牙是一种支持设备短距离通信（一般是10m以内）的无线技术，其数据传输时不仅不需要连线，而且传输速率也比传统手持设备的红外模式更加迅速、高效，主要优势如下所列。

- 免费。

蓝牙无线技术规格供全球的成员公司免费使用。除了设备费用外，制造商不需要为使用蓝牙技术再支付任何知识产权费用，这大大降低了蓝牙技术的普及门槛。

- 应用范围广。

蓝牙技术得到了广泛的应用，集成该技术的产品从手机、汽车到医疗设备等应有尽有，使用该技术的用户从消费者、工业市场到企业等，不一而足。

- 易于使用。

蓝牙是一项即时技术，其不要求固定的基础设施，且易于安装和设置，而且无须电缆即可实现连接，使用起来非常方便。

> **提示**
>
> 上一小节介绍的Socket技术虽然开发很容易，但对用户而言要么需要使用3G网络，要么需要通过Wifi网络。3G网络虽然没有位置限制，但费用不低；Wifi虽然免费，但必须在AP附近或自己架设AP。可以看出，对于短距离即时互联而言，采用蓝牙技术更为合理、便捷。

- 全球通用的规格。

蓝牙无线技术是当今市场上支持范围最广泛，功能最丰富且安全的无线标准之一，全球范围内的资格认证程序可以测试成员的产品是否符合标准。

介绍完了蓝牙技术的特点与优势后，下面简单介绍一下蓝牙设备的使用步骤，具体如下所列。

（1）开启要搜索的设备的蓝牙功能，并设置为可见。
（2）在一个设备中开启搜索设备的功能，开始搜索设备。
（3）当搜索到其他设备后，会将搜索的设备显示在本设备的列表中。
（4）选择列表中的某一设备，请求匹配。
（5）被选中的设备收到匹配请求后，经双方验证同意，设备匹配成功。
（6）设备匹配成功后就可以建立连接，并收发数据了。

> **提示**
>
> 蓝牙通信与Socket网络通信的基本思想非常类似，都是连接成功后建立双向数据流收发数据，但开发起来要比Socket复杂一些。这主要是因为蓝牙设备的搜索功能和配对列表的显示需要开发人员自行编写代码实现，下面几个小节的案例将对此进行详细的介绍。

2.7.2　聊天案例概览

本小节将介绍一个用蓝牙技术实现聊天功能的案例，通过对本案例的学习，读者可以掌握关于 蓝牙通信的整个开发过程。在开发具体代码之前，首先了解一下本案例的运行方法及运行效果，具体情况如下所列。

（1）准备两部Android手机，在两部手机上都安装本案例的apk（Sample2_10.apk）。

> **提示**
>
> 笔者使用的两部手机分别为华为u8800和摩托罗拉ME525，其他支持蓝牙功能的Android智能手机也可以使用。之所以要使用真机是因为模拟器几乎无法进行蓝牙程序的模拟测试，真机则要方便得多。

（2）将两部手机的蓝牙功能都打开，并设置为可见。
（3）在两部手机中同时运行本案例。
（4）在其中的一部手机中单击Menu键（手机上自带的菜单键）弹出设备搜索列表，如图2-31所示。然后单击列表下方的“扫描设备”按钮搜索设备，搜索完毕后列表中出现另一手机的设备名称及硬件地址，如图2-32所示。



图2-31　在u8800中弹出设备搜索列表

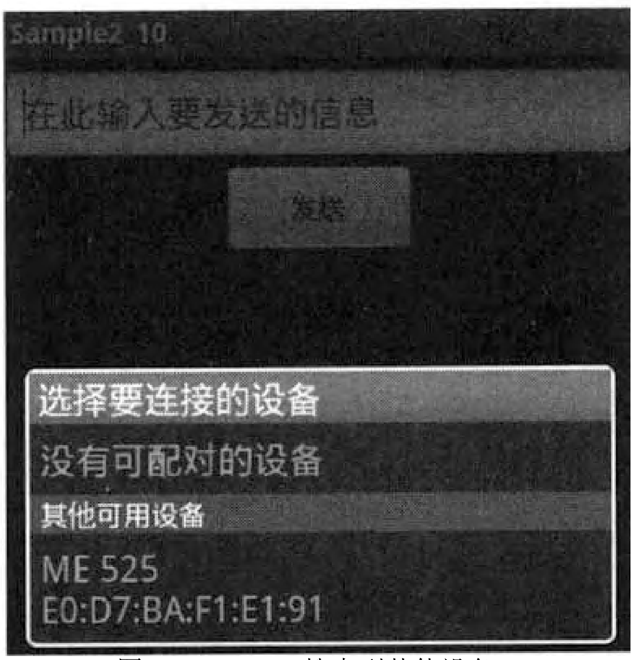


图2-32　u8800搜索到其他设备

（5）单击列表中搜索到的设备名称，一段时间后两部手机会同时提示连接成功，如图2-33和图2-34所示。



图2-33　u8800提示已连接到ME 525



图2-34　ME 525提示已连接到u8800

（6）在一部手机聊天界面的文本框中输入一些文本信息，再单击“发送”按钮，另一部手机会弹出该手机的名称及接收到的消息，如图2-35和图2-36所示。

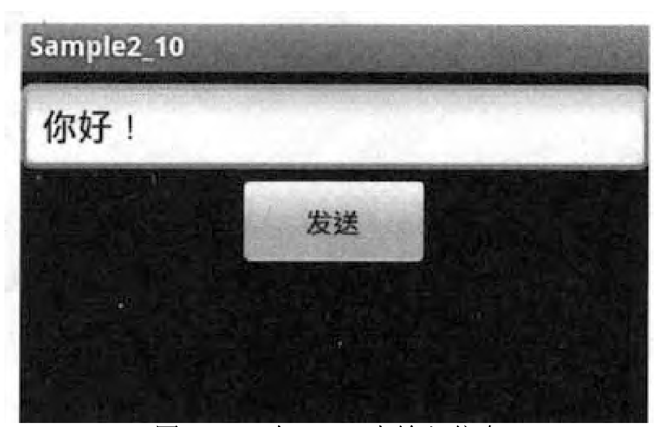


图2-35　在u8800中输入信息



图2-36　ME 525收到由u8800发送来的消息

提示

本案例运行时的界面还有很多，由于篇幅所限，这里只给出主要的运行界面，其他界面请读者自行运行随书光盘中的本案例进行测试。

2.7.3　聊天案例的开发过程

介绍完本案例的运行过程及效果后，本小节将详细介绍本案例的开发过程。由于本案例中涉及的类比较多，因此在开发代码之前首先介绍一下本案例中各个类之间的关系及各自的用途，如图2-37所示。

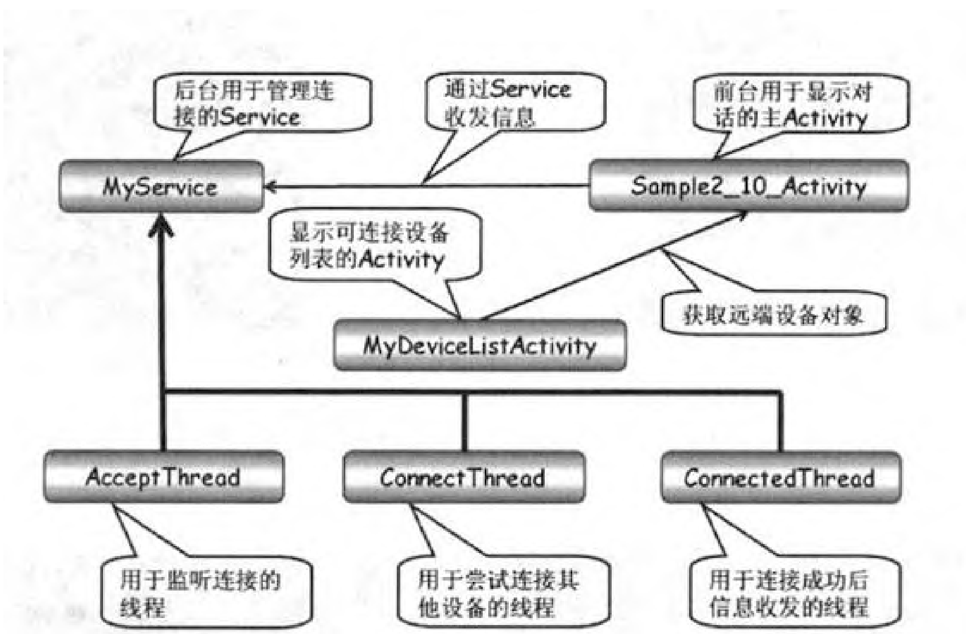


图2-37　案例中各个类之间的关系及各自的用途

从图2-37中可以看出，本案例中包含两个Activity、一个Service、3个线程，各自的用途如下所列。

- Sample2_10_Activity为本案例的主Activity，程序一启动其就开始运行，主要功能为用来显示对话信息；MyDeviceListActivity用于显示搜索到的可连接设备列表，通过菜单启动；MyService则用来在后台管理蓝牙连接。
- AcceptThread用于在蓝牙连接请求接收方监听连接请求；ConnectThread用来向别的设备发出连接请求；ConnectedThread用来在蓝牙连接建立后接收对方设备发送过来的消息。

了解完本案例中各个类之间的关系及各自的用途后，就可以进行代码的开发了，具体步骤如下 所列。

（1）首先开发本案例的主控制类——Sample2_10_Activity，该类在程序开始时执行，其中重写了onCreate方法、onStart方法以及onDestroy等方法，具体代码如下。



代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10目录下的Sample2_10_Activity.java。

```java
package com.bn.pp10;                                          //声明包
import android.app.Activity;                                  //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.widget.Toast;                                  //引入相关类
public class Sample2_10_Activity extends Activity {
    private EditText outEt;                                   // 布局中的控件引用
    private Button sendBtn;
    private String connectedNameStr = null;                   // 已连接的设备名称
    private StringBuffer outSb;                               // 发送的字符信息
        private BluetoothAdapter btAdapter = null;            // 本地蓝牙适配器
        private MyService myService = null;                   // Service 引用
        public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);
            btAdapter = BluetoothAdapter.getDefaultAdapter();// 获取本地蓝牙适配器
        }
        public void onStart() {
            super.onStart();
                                    // 如果蓝牙没有开启，提示开启蓝牙，并退出 Activity
            if (!btAdapter.isEnabled()) {
                Toast.makeText(this, "请先开启蓝牙！", Toast.LENGTH_LONG).show();
                finish();
            } else {                                          // 否则初始化聊天的控件
                if (myService == null)
                    initChat();
        }}
        public synchronized void onResume() {
            super.onResume();
            if (myService != null) {                          // 创建并开启 Service
                if (myService.getState() == MyService.STATE_NONE) {// 如果 Service 为空状态
                    myService.start();                        // 开启 Service
        }}}
        private void initChat() {
            outEt = (EditText) findViewById(R.id.edit_text_out); // 获取编辑文本框的引用
            sendBtn = (Button) findViewById(R.id.button_send);
                                          // 获取发送按钮引用，并为其添加监听
            sendBtn.setOnClickListener(new OnClickListener() {
                public void onClick(View v) {
                                          // 获取编辑文本框中的文本内容，并发送消息
                    TextView view = (TextView) findViewById(R.id.edit_text_out);
                    String message = view.getText().toString();
                    sendMessage(message);
            }});
```

```
43            myService = new MyService(this, mHandler);          // 创建 Service 对象
44            outSb = new StringBuffer("");          // 初始化存储发送消息的 StringBuffer
45        }
46        public void onDestroy() {
47            super.onDestroy();
48            if (myService != null) {                             // 停止 Service
49                myService.stop();
50        }}
51        private void sendMessage(String message) {               // 发送消息的方法
52            ……//此处省略了部分源代码，将在后面步骤中给出
53        }}
54                                                   //处理从 Service 发来消息的 Handler
55        private final Handler mHandler = new Handler() {   .
56            ……//此处省略了部分源代码，将在后面步骤中给出
57        }}};
58        public void onActivityResult(int requestCode, int resultCode, Intent data) {
59            ……//此处省略了部分源代码，将在后面步骤中给出
60        }}
61        public boolean onPrepareOptionsMenu(Menu menu) {
62                                                   // 启动设备列表 Activity 搜索设备
63            Intent serverIntent = new Intent(this, MyDeviceListActivity.class);
64            startActivityForResult(serverIntent, 1);
65            return true;
66    }}
```

- 第6-16行是该类成员变量的定义及重写的onCreate方法，其主要工作为跳转到主界面并通过BluetoothAdapter类的静态方法getDefaultAdapter获取蓝牙适配器对象的引用。
- 第17-26行重写了onStart方法。在该方法中查看蓝牙是否开启，如果蓝牙没有开启，则提示开启蓝牙，并退出Activity；否则调用initChat方法初始化聊天界面。
- 第27-32行重写了onResume方法。在该方法中检测是否已经启动了后台服务，若后台服务为空，则调用其start方法，开启后台服务。此后台服务用于管理蓝牙的连接及数据的收发，具体开发在后面进行介绍。
- 第33-45行为初始化聊天界面的方法。其中首先为发送按钮添加了监听器，接着创建了后台服务对象与字符串缓冲对象。
- 第51-60行省略了发送消息的sendMessage方法、onActivityResult方法及Handler内部类的代码，这些省略部分将在后面一一进行详细介绍。
- 第61-66行为每次按下Menu键调用的方法。在该方法中创建一个Intent消息，通过其启动搜索设备列表的Activity。其中的“1”与onActivityResult方法中的

“1”相对应。

（2）接下来介绍Sample2_10_Activity类中省略的sendMessage方法。该方法负责发送消息，其代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10目录下的Sample2_10_Activity.java。

```
1private void sendMessage(String message) {          // 发送消息的方法
2                                                    // 先检查是否已经连接到设备
3    if (myService.getState() != MyService.STATE_CONNECTED) {
4        Toast.makeText(this, R.string.not_connected, Toast.LENGTH_SHORT)
5                .show();
6        return;
7    }
8    if (message.length() > 0) {                     // 如果消息不为空再发送消息
9        byte[] send = message.getBytes();           // 获取发送消息的字节数组，并发送
10           myService.write(send);
11           outSb.setLength(0);                     //消除 StringBuffer 和文本框的内容
12           outEt.setText(outSb);
13   }};
```

• 第1-7行检查是否已成功经连接到设备，如果没有连接成功，则弹出提示信息。

• 第8-12行为在连接成功的情况下，判断待发送的消息是否为空，若消息不为空，则再将信息以字节数组的形式，通过myService的write方法发送出去。

（3）然后介绍前面省略的处理从Service发来的消息的Handler类的开发，用下列代码替代前面Sample2_10_Activity类中的第55-57行代码。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10目录下的Sample2_10_Activity.java。

```
1// 处理从Service发来的消息的Handler
2private final Handler mHandler = new Handler() {
3    public void handleMessage(Message msg) {
4        switch (msg.what) {
5        case Constant.MSG_READ:
6            byte[] readBuf = (byte[]) msg.obj;
7            String readMessage = new String(readBuf, 0, msg.arg1);//创建要发送信息的字符串
8            Toast.makeText(Sample2_10_Activity.this,
9                    connectedNameStr + ":  " + readMessage,
10                       Toast.LENGTH_LONG).show();
11               break;
12           case Constant.MSG_DEVICE_NAME:
13                                              // 获取已连接的设备名称
14               connectedNameStr = msg.getData().getString(
15                       Constant.DEVICE_NAME);
16               Toast.makeText(getApplicationContext(),     //并弹出提示信息
17                       "已连接到 " + connectedNameStr, Toast.LENGTH_SHORT)
18                       .show();
19               break;
20   }}};
```

- 第4-11行为收到待读数据信息时的处理代码，其功能为从msg中获取字节数组形式的信息，将其转化成字符串，并以Toast的形式显示在屏幕上。
- 第12-19行为收到已连接设备名称信息的处理代码，其功能为获取已连接设备的名称，并弹出Toast提示信息。

（4）接下来开发重写的onActivityResult方法，用下列代码替代前面Sample2_10_Activity类中的第58-60行代码。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10目录下的Sample2_10_Activity.java。

```
1 public void onActivityResult(int requestCode, int resultCode, Intent data) {
2     switch (requestCode) {
3     case 1:                              // 如果设备列表Activity返回一个连接的设备
4         if (resultCode == Activity.RESULT_OK) {
5             String address = data.getExtras().getString(// 获取设备的MAC地址
6                     MyDeviceListActivity.EXTRA_DEVICE_ADDR);
7             BluetoothDevice device = btAdapter
8                     .getRemoteDevice(address);// 获取BLuetoothDevice对象
9             myService.connect(device);        // 连接该设备
10            }
11            break;
12    }};
```

• 第1行为该方法的签名，其中requestCode参数用来标识是从哪一个Activity跳转到该Activity，resultCode参数用来表示返回值的状态，data参数里包含了返回的数据。

• 第2-12行功能为：若设备列表Activity成功返回了选中的目标连接设备，则调用蓝牙适配器对象的getRemoteDevice方法获取远端设备对象，然后将远端设备对象传递给自定义的后台服务连接该设备。

（5）下一步将开发用于显示可连接设备列表的Activity类——MyDeviceListActivity类，其代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10目录下的MyDeviceListActivity.java。

```java
package com.bn.pp10;                                                    //声明包
import java.util.Set;                                                   //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.widget.AdapterView.OnItemClickListener;                  //引入相关类
public class MyDeviceListActivity extends Activity {
    public static String EXTRA_DEVICE_ADDR = "device_address";         // extra 信息名称
    private BluetoothAdapter myBtAdapter;                               // 成员变量
    private ArrayAdapter<String> myAdapterPaired;
    private ArrayAdapter<String> myAdapterNew;
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            requestWindowFeature(Window.FEATURE_INDETERMINATE_PROGRESS);// 设置窗口
            setContentView(R.layout.device_list);
            // 设置为当结果是 Activity.RESULT_CANCELED 时，返回到该 Activity 的调用者
            setResult(Activity.RESULT_CANCELED);
            Button scanBtn = (Button) findViewById(R.id.button_scan);// 初始化搜索按钮
            scanBtn.setOnClickListener(new OnClickListener() {
                public void onClick(View v) {
                    doDiscovery();
                    v.setVisibility(View.GONE);                         // 使按钮不可见
                }
            });
                                                                        // 初始化适配器
```

```
24            myAdapterPaired = new ArrayAdapter<String>(this,
25                    R.layout.device_name);                    // 已配对的
26            myAdapterNew = new ArrayAdapter<String>(this,
27                    R.layout.device_name);                    // 新发现的
28                                                              // 将已配对的设备放入列表中
29            ListView lvPaired = (ListView) findViewById(R.id.paired_devices);
30            lvPaired.setAdapter(myAdapterPaired);
31            lvPaired.setOnItemClickListener(mDeviceClickListener);
32                                                              // 将新发现的设备放入列表中
33            ListView lvNewDevices = (ListView) findViewById(R.id.new_devices);
34            lvNewDevices.setAdapter(myAdapterNew);
35            lvNewDevices.setOnItemClickListener(mDeviceClickListener);
36                                                              // 注册发现设备时的广播
37            IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
38            this.registerReceiver(mReceiver, filter);
39                                                              // 注册搜索完成时的广播
40            filter = new IntentFilter(BluetoothAdapter.ACTION_DISCOVERY_FINISHED);
41            this.registerReceiver(mReceiver, filter);
42            myBtAdapter = BluetoothAdapter.getDefaultAdapter(); // 获取本地蓝牙适配器
43        Set<BluetoothDevice> pairedDevices = myBtAdapter.getBondedDevices();
                                                                // 获取已配对的设备
44                                                      // 将所有已配对设备信息放入列表中
45            if (pairedDevices.size() > 0) {
46                findViewById(R.id.title_paired_devices).setVisibility(View.VISIBLE);
47                for (BluetoothDevice device : pairedDevices) {
48                    myAdapterPaired.add(device.getName() + "\n"
49                            + device.getAddress());
50                }
51            } else {
52                String noDevices = getResources().getText(R.string.none_paired)
53                        .toString();
54                myAdapterPaired.add(noDevices);
55        }}
56        protected void onDestroy() {
57            ……//此处省略了部分源代码，该方法非常简单，请读者自行查看随书光盘中的源代码。
58        }
59                                                      //用蓝牙适配器搜索设备的方法
60        private void doDiscovery() {
61            ……//此处省略了部分源代码，将在后面步骤中给出
62        }
63                                                      // 列表中设备按下时的监听器
64        private OnItemClickListener mDeviceClickListener = new OnItemClickListener()
{
65            ……//此处省略了部分源代码，将在后面步骤中给出
66        };
67                                                  // 监听搜索到的设备的 BroadcastReceiver
68        private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
69            ……//此处省略了部分源代码，将在后面步骤中给出
70    };}
```

- 第6-9行为该类成员变量的定义。其中有extra信息的名称、蓝牙适配器对象的引用、已配对设备和新搜索到设备的列表对应适配器的声明。
- 第10-55行为重写的onCreate方法，其中首先初始化了成员变量及布局文件中的相关资源，然后注册发现设备和搜索完成时的广播接收器，并初始化了已配对设备列表。
- 第60-70行省略了doDiscovery方法、mDeviceClickListener监听器及mReceiver内部类的具体代码，这些代码将在后面一一进行详细介绍。

提示

由于篇幅所限，设备列表布局文件device_list.xml代码的开发在此没有进行介绍，需要的读者请自行查看随书光盘中的源代码。

（6）接下来将开发前面省略的负责搜索设备的doDiscovery方法，其代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10目录下的MyDeviceListActivity.java。

```
private void doDiscovery() {
                                              // 在标题上显示正在搜索的标志
    setProgressBarIndeterminateVisibility(true);
    setTitle(R.string.scanning);
                                              // 显示搜索到的新设备的副标题
    findViewById(R.id.title_new_devices).setVisibility(View.VISIBLE);
    if (myBtAdapter.isDiscovering()) {        // 如果正在搜索，取消本次搜索
        myBtAdapter.cancelDiscovery();
    }
        myBtAdapter.startDiscovery();         // 开始搜索
};
```

说明

在该方法中首先更改UI界面（如在标题上显示正在搜索的标志），完成相关操作后，调用蓝牙适配器的startDiscovery方法进行设备搜索。

（7）下一步将开发前面省略的监听器mDeviceClickListener的对应代码，具体内容如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10目录下的MyDeviceListActivity.java。

```
1 private OnItemClickListener mDeviceClickListener = new OnItemClickListener() {
2     public void onItemClick(AdapterView<?> av, View v, int arg2, long arg3) {
3         myBtAdapter.cancelDiscovery();                    // 取消搜索
4         String msg = ((TextView) v).getText().toString();   // 获取设备的MAC地址
5         String address = msg.substring(msg.length() - 17);
6         Intent intent = new Intent();                     // 创建带有MAC地址的Intent
7         intent.putExtra(EXTRA_DEVICE_ADDR, address);
8         setResult(Activity.RESULT_OK, intent);            // 设置结果并退出Activity
9         finish();
10        }};
```

- 上述实现监听功能的onltemClick方法中，首先取消搜索，然后获取并发送设备的MAC地址给主界面Activity。
- 第8行setResult方法中的参数Activity.RESULT_OK表示成功地获取了要连接的设备的硬件地址。接收方的Activity根据Activity.RESULT_OK会做出相应的处理，前面步骤4中已经进行了 介绍。

（8）接下来将开发前面省略的用于接收设备搜索到系统广播的广播接收器——mReceiver，其代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10

目录下的MyDeviceListActivity.java。

```
1                                               // 监听搜索到的设备的BroadcastReceiver
2private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
3    @Override
4    public void onReceive(Context context, Intent intent) {
5        String action = intent.getAction();
6        if (BluetoothDevice.ACTION_FOUND.equals(action)) {// 如果找到设备
7                                               // 从Intent中获取BluetoothDevice对象
8            BluetoothDevice device = intent
9                    .getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
10                                              // 如果没有配对，将设备加入新设备列表
11               if (device.getBondState() != BluetoothDevice.BOND_BONDED) {
12                   myAdapterNew.add(device.getName() + "\n"
13                           + device.getAddress());
14               }
15                                              // 当搜索完成后，改变Activity的标题
16           } else if (BluetoothAdapter.ACTION_DISCOVERY_FINISHED
17                   .equals(action)) {
18               setProgressBarIndeterminateVisibility(false);
19               setTitle(R.string.select_device);
20               if (myAdapterNew.getCount() == 0) {        //没有找到设备
21                   String noDevices = getResources().getText(
22                           R.string.none_found).toString();
23                   myAdapterNew.add(noDevices);
24               }
25   }}};
```

- 第6-14行为此广播接收器收到找到设备信息时对应的处理代码。首先从Intent中获取BluetoothDevice对象，然后判断该设备是否已经配对。若没有配对，则将设备加入新设备列表，否则不进行任何处理。

第16-24行为此广播接收器收到完成设备搜索信息时对应的处理代码，主要是改变对话框的标题等信息。如果没有找到任何设备，则在界面中显示没有找到设备的信息。

（9）再接着将开发自定义的用于后台服务的MyService类，其提供的后台服务主要是关于蓝牙设备的连接、数据的收发等方面，具体代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10目录下的MyService.java。

```
package com.bn.pp10;                                    //声明包
import java.io.IOException;                             //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.os.Message;                              //引入相关类
public class MyService {                                //用于管理连接的Service
                                                        // 本应用的唯一 UUID
```

```java
    private static final UUID MY_UUID =
        UUID.fromString("fa87c0d0-afac-11de-8a39-0800200c9a66");
    private final BluetoothAdapter btAdapter;            // 成员变量
    private final Handler myHandler;
    private AcceptThread myAcceptThread;
    private ConnectThread myConnectThread;
    private ConnectedThread myConnectedThread;
    private int myState;
                                                          // 表示当前连接状态的常量
    public static final int STATE_NONE = 0;              // 什么也没做
    public static final int STATE_LISTEN = 1;            // 正在监听连接
    public static final int STATE_CONNECTING = 2;        // 正在连接
    public static final int STATE_CONNECTED = 3;         // 已连接到设备
                                                          // 构造器
    public MyService(Context context, Handler handler) {
        btAdapter = BluetoothAdapter.getDefaultAdapter();
        myState = STATE_NONE;
        myHandler = handler;
    }
    private synchronized void setState(int state) {  //设置当前连接状态的方法
        myState = state;
    }
    public synchronized int getState() {              //获取当前连接状态的方法
        return myState;
    }
    public synchronized void start() {                //开启 Service 的方法
                                                       // 关闭不必要的线程
        if (myConnectThread != null) {myConnectThread.cancel(); myConnectThread = null;}
        if (myConnectedThread != null) {myConnectedThread.cancel(); myConnectedThread
        = null;}
        if (myAcceptThread == null) {                 // 开启线程监听连接
            myAcceptThread = new AcceptThread();
            myAcceptThread.start();
        }
        setState(STATE_LISTEN);
    }
    public synchronized void stop() {                 //停止所有线程的方法
        if (myConnectThread != null) {myConnectThread.cancel(); myConnectThread =
        null;}
        if (myConnectedThread != null) {myConnectedThread.cancel(); myConnectedThread
        = null;}
        if (myAcceptThread != null) {myAcceptThread.cancel(); myAcceptThread = null;}
        setState(STATE_NONE);
    }
    public void write(byte[] out) {              //向 ConnectedThread 写入数据的方法
        ConnectedThread tmpCt;                   // 创建临时对象引用
        synchronized (this) {                    // 锁定 ConnectedThread
            if (myState != STATE_CONNECTED) return;
            tmpCt = myConnectedThread;
        }
        tmpCt.write(out);                        // 写入数据
    }
```

```
56      public synchronized void connect(BluetoothDevice device) {//连接设备的方法
57        ……//此处省略了部分源代码，将在后面步骤中给出
58      }
59                                              //开启管理和已连接的设备间通话的线程的方法
60      public synchronized void connected(BluetoothSocket socket, BluetoothDevice
device) {
61        ……//此处省略了部分源代码，将在后面步骤中给出
62      }
63      ……//以下省略了3个主要线程类的代码，将在后面介绍
64  }
```

- 第6-19行为成员变量的定义，其中第7行定义了本应用的UUID（全局唯一标识）。第16-19行定义了表示当前连接状态的一些常量，各常量代表的含义详见代码注释。
- 第26-31行为设置和获取当前连接状态的方法，这两个方法都是用synchronized修饰符修饰的，这是为了避免异步线程可能同时读取myState时产生的问题。
- 第32-41行为开启后台服务的方法。在该方法中首先关闭不必要的线程，然后开启线程监听连接，并设置当前状态为正在监听。
- 第42-47行为停止所有线程的stop方法，主要工作是调用各线程的cancel方法。• 第48-55行为向连接线程ConnectedThread写入数据的write方法，只要调用ConnectedThread对象的write方法即可。

> **提示**
>
> 第56-63行省略了connect方法、connected方法及3个主要线程类的代码，这些代码的开发将在下面进行详细讲解。

（10）接下来将开发负责连接设备的connect方法，其代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10目录下的MyService.java。

```
public synchronized void connect(BluetoothDevice device) {//连接设备的方法
                                                    // 关闭不必要的线程
    if (myState == STATE_CONNECTING) {
        if (myConnectThread != null) {myConnectThread.cancel(); myConnectThread = null;}
    }
    if (myConnectedThread != null) {myConnectedThread.cancel(); myConnectedThread = null;}
    myConnectThread = new ConnectThread(device);  // 开启线程连接设备
    myConnectThread.start();
    setState(STATE_CONNECTING);
    }
```

说明

该方法的实现比较简单，首先关闭不必要的线程，然后开启线程连接设备，并设置当前状态为正在连接。

（11）接着将开发负责开启连接线程（ConnectedThread）并向myHandler发送设备名称消息的connected方法，其代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10目录下的MyService.java。

```
public synchronized void connected(BluetoothSocket socket, BluetoothDevice device)
{
                                    // 关闭不必要的线程
    if (myConnectThread != null) {myConnectThread.cancel(); myConnectThread = null;}
   if (myConnectedThread != null) {myConnectedThread.cancel(); myConnectedThread =
null;}
    if (myAcceptThread != null) {myAcceptThread.cancel(); myAcceptThread = null;}
    myConnectedThread = new ConnectedThread(socket); // 创建并启动ConnectedThread
    myConnectedThread.start();
                                    // 发送已连接的设备名称到主界面Activity
    Message msg = myHandler.obtainMessage(Constant.MSG_DEVICE_NAME);
        Bundle bundle = new Bundle();
        bundle.putString(Constant.DEVICE_NAME, device.getName());
        msg.setData(bundle);
        myHandler.sendMessage(msg);
        setState(STATE_CONNECTED);
    }
```

> **提示**
>
> 在该方法中首先要关闭不必要的线程，然后创建并开启ConnectedThread线程，同时发送已连接的设备名称到主界面Activity，最后设置当前状态为已连接。

（12）接下来将开发前面省略的MyService类中的3个主要线程，首先开发负责监听连接请求的AcceptThread，其代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10目录下的MyService.java。

```
private class AcceptThread extends Thread {      //用于监听连接的线程
                                                 // 本地服务器端 ServerSocket
   private final BluetoothServerSocket mmServerSocket;
   public AcceptThread() {
      BluetoothServerSocket tmpSS = null;
      try {                               // 创建用于监听的服务器端 ServerSocket
   tmpSS = btAdapter.listenUsingRfcommWithServiceRecord("BluetoothChat", MY_UUID);
      } catch (IOException e) {e.printStackTrace();}
      mmServerSocket = tmpSS;
       }
       public void run() {
          setName("AcceptThread");                   //设置线程名称
          BluetoothSocket socket = null;
          while (myState != STATE_CONNECTED) {       //如果没有连接到设备
             try {
                socket = mmServerSocket.accept();  //获取连接的 Socket
             } catch (IOException e) {e.printStackTrace();break;}
             if (socket != null) {                   // 如果连接成功
                synchronized (MyService.this) {
                   switch (myState) {
                   case STATE_LISTEN:
                   case STATE_CONNECTING:
                                                     // 开启管理连接后数据交流的线程
                      connected(socket, socket.getRemoteDevice());
```

```
25                     break;
26                 case STATE_NONE:
27                 case STATE_CONNECTED:
28                     try {                    // 关闭新Socket
29                         socket.close();
30                     } catch (IOException e) {
31                        e.printStackTrace();
32                     }
33                     break;
34         }}}}}
35         public void cancel() {//关闭本地服务器端ServerSocket的方法
36             try {
37                 mmServerSocket.close();   //调用close方法关闭ServerSocket
38             } catch (IOException e) {e.printStackTrace();}
39     }}
```

- 第3-10行为该线程类的成员变量及构造器声明，在构造器中创建了用于在服务端监听并接收连接请求的BluetoothServerSocket对象。

第11-34行为重写的用于描述线程任务的run方法。在该方法的while循环中，如果没有连接到设备，则一直调用BluetoothServerSocket的accept方法进入阻塞状态，等待连接请求的到来。

（13）下面将开发负责用于尝试连接其他设备的ConnectThread，其代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10目录下的MyService.java。

```
private class ConnectThread extends Thread {      //用于尝试连接其他设备的线程
   private final BluetoothSocket myBtSocket;
   private final BluetoothDevice mmDevice;
   public ConnectThread(BluetoothDevice device) {
      mmDevice = device;
      BluetoothSocket tmp = null;
                                        // 通过正在连接的设备获取BluetoothSocket
      try {
         tmp = device.createRfcommSocketToServiceRecord(MY_UUID);
         } catch (IOException e) {e.printStackTrace();}
         myBtSocket = tmp;
      }
      public void run() {
         setName("ConnectThread");
         btAdapter.cancelDiscovery();              // 取消搜索设备
         try {                                     // 连接到BluetoothSocket
            myBtSocket.connect();                  //尝试连接
         } catch (IOException e) {
            setState(STATE_LISTEN);                //连接断开后设置状态为正在监听
            try {                                  // 关闭Socket
               myBtSocket.close();
            } catch (IOException e2) {e.printStackTrace();}
            MyService.this.start();                //如果连接不成功，重新开启Service
            return;
         }
         synchronized (MyService.this) {           // 将ConnectThread线程置空
            myConnectThread = null;
         }
         connected(myBtSocket, mmDevice);               // 开启管理连接后数据交流的线程
      }
      public void cancel() {
         try {
            myBtSocket.close();
         } catch (IOException e) {e.printStackTrace();}
      } }
```

- 第2-12行为该线程类的成员变量及构造器的声明。在构造器中通过正在连接的设备获取BluetoothSocket对象的引用。
- 第13-30行为重写的run方法，在该方法中主要调用BluetoothServerSocket类的connect方法尝试连接。如果连接被断开，改变相应的状态，并释放资源。

（14）接下来将开发负责连接成功后信息收发的ConnectedThread，其代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10/src/com/bn/pp10目录下的MyService.java。

```
1   private class ConnectedThread extends Thread {
2       private final BluetoothSocket myBtSocket;
3       private final InputStream mmInStream;
4       private final OutputStream myOs;
5       public ConnectedThread(BluetoothSocket socket) {
6           myBtSocket = socket;
7           InputStream tmpIn = null;
8           OutputStream tmpOut = null;
9           try {                                   //获取BluetoothSocket的输入输出流
10                  tmpIn = socket.getInputStream();
11                  tmpOut = socket.getOutputStream();
12              } catch (IOException e) {e.printStackTrace();} //打印异常
13              mmInStream = tmpIn;
14              myOs = tmpOut;
15          }
16          public void run() {
17              byte[] buffer = new byte[1024];
18              int bytes;
19              while (true) {                      // 一直监听输入流
20                  try {
21                      bytes = mmInStream.read(buffer);// 从输入流中读入数据
22                                                  //将读入的数据发送到主界面Activity
23                      myHandler.obtainMessage(Constant.MSG_READ, bytes, -1, buffer)
24                              .sendToTarget();
25                  } catch (IOException e) {
26                      e.printStackTrace();
27                      setState(STATE_LISTEN);     //连接断开后设置状态为正在监听
28                      break;
29          }}}
30          public void write(byte[] buffer) {      //向输出流中写入数据的方法
31              try {
32                  myOs.write(buffer);
33              } catch (IOException e) {e.printStackTrace();}
34          }
35          public void cancel() {
36              try {
37                  myBtSocket.close();                     //关闭Socket
38              } catch (IOException e) {e.printStackTrace();}
39          }}
```

- 第2-15行为该线程类的成员变量及构造器的声明。在构造器中通过传递进来的BluetoothSocket对象的引用获取基于蓝牙连接的输入输出流。

- 第16-29行为重写的run方法。在该方法中通过while循环一直监听输入流，一旦输入流中有数据，便从输入流中读出数据，同时将数据发送到主界面Activity。
- 第30-34行为向输出流中写入数据的方法，该方法只是对输出流对象的write方法进行了简单的封装。

（15）代码开发完成后还需要在AndroidManifest.xml中声明BLUETOOTH权限，其代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10目录下的AndroidManifest.xml。

```
<!--声明BLUETOOTH权限-->
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.BLUETOOTH" />
```

> **提示**
>
> 上述代码应插入到AndroidManifest.xml中的“</manifest>”标签之前。

（16）在AndroidManifest.xml文件中声明完BLUETOOTH权限后，接下来需要在AndroidManifest.xml文件中注册自己开发的MyDeviceListActivity，具体代码如下。

代码位置：见本书随书光盘中源代码/第2章/Sample2_10目录下的AndroidManifest.xml。

```
<!--注册MyDeviceListActivity-->
<activity android:name=".MyDeviceListActivity"
      android:label="@string/select_device"
      android:theme="@android:style/Theme.Dialog"
      android:configChanges="orientation|keyboardHidden"
/>
```

> **提示**
>
> 上述代码应插入到AndroidManifest.xml中的<application>与<application>标签之间。请读者注意的是，在新建项目时指定的Activity其配置代码系统会自动加入到AndroidManifest.xml文件中，但自己在项目中再开发其他的Activity时就需要自行添加相应的配置代码到AndroidManifest.xml文件中了。

2.8　本章小结

本章主要对Android平台下游戏开发中可能会用到的Android平台的相关基础知识进行了详细介绍，如文件的I/O操作、SQLite数据库的增、删、改、查，游戏中声音特效、蓝牙互联等。只有学好这些基础知识，开发人员才能开发出更好的Android手机游戏。读者一定要在这些基础上面下足了工夫，今后的开发道路才会一帆风顺。

提示

Android平台的其他相关知识还有很多，由于本书侧重于介绍OpenGL ES 2.0 3D应用程序的开发，故本章只是简单介绍了一些与3D开发不直接相关的，但在游戏开发中非常重要的Android平台相关的知识。若读者对Android平台很不熟悉，可以首先参考笔者在人民邮电出版社出版的《Android应用开发完全自学手册——核心技术、传感器、2D/3D、多媒体与典型案例》一书，其对Android各方面的基础知识都进行了详细的介绍。

第3章 初识OpenGL ES 2.0

OpenGL ES是基于OpenGL三维图形API的子集，主要是针对于手机以及PDA（掌上电脑）等嵌入式设备设计的。OpenGL ES的API由Khronos组织定义并推广，该组织致力于为这些API建立免费的开发标准。

本章主要对OpenGL ES 2.0进行简要介绍，为读者介绍学习OpenGL ES 2.0需要了解的部分知识点。包括OpenGL ES 2.0简介、初识OpenGL ES 2.0应用程序以及OpenGL ES 2.0与1.x的渲染管线等。

3.1 OpenGL ES 2.0概览

随着3G时代的到来，Android与iPhone逐渐成为消费者购买智能手机的主要选择。而由于基于Android的智能手机性能优良、价格合适，因此Android智能手机得到了大多数用户的青睐。

随着Android系统版本及硬件水平的提升，OpenGL ES版本也由原先仅支持固定渲染管线的OpenGL ES 1.x升级为支持自定义渲染管线的OpenGL ES 2.0。这使得使用OpenGL ES 2.0渲染的3D场景更加真实，从而能够创造全新的用户体验。

3.1.1 OpenGL ES 2.0简介

现今较为知名的3D图形API有OpenGL、DirectX以及OpenGL ES，它们各自的应用领域如下。

- DirectX主要应用于Windows下游戏的开发，在此领域基本上一统天下。
- OpenGL的应用领域较为广泛，适用于UNIX、Mac OS、Linux以及Microsoft等几乎所有的操作系统，可以开发游戏、工业建模以及嵌入式设备。
- OpenGL ES是专门针对于嵌入式设备而设计的，其实际是OpenGL的剪裁版本，去除了OpenGL中许多不是必须存在的特性，如：GL_QUADS（四边形）与GL_POLYGONS（多边形）绘制模式以及glBegin（开始）/glEnd（结束）操作等。

经过多年的发展，OpenGL ES主要分为两个主版本，基本情况如下。

- 一个是OpenGL ES 1.x（主要包括1.0与1.1），其采用的是固定渲染管线，可以由硬件GPU支持或用软件模拟实现，渲染能力有限，在纯软件模拟情况下性能也较弱。如图3-1所示的都市赛车5就是使用OpenGL ES 1.x渲染技术的。



图3-1　都市赛车5场景

• 另一个是OpenGL ES 2.0，其采用的是可编程渲染管线，渲染能力大大提高。OpenGL ES 2.0 要求设备中必须有相应的GPU硬件支持，目前不支持在设备上用软件模拟实现。如图3-2所示的都市赛车6就是使用OpenGL ES 2.0渲染技术的。

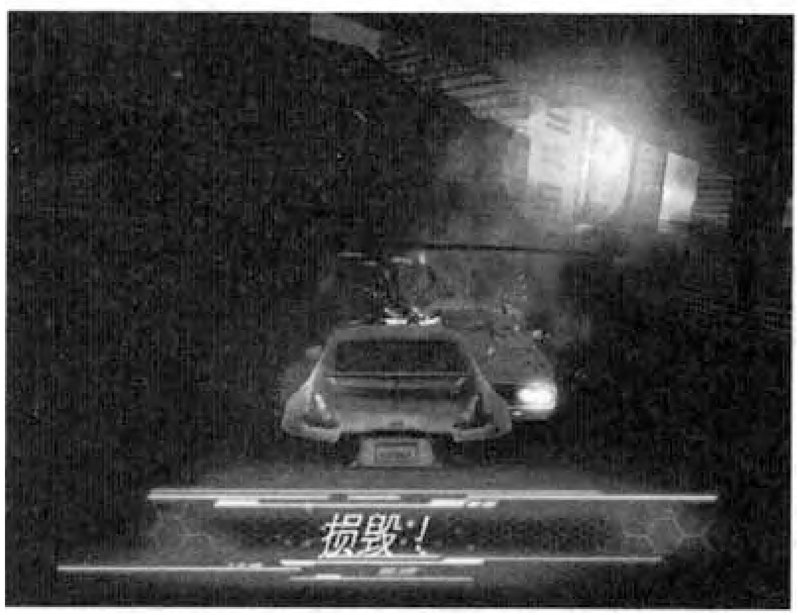


图3-2　都市赛车6场景

说明

图3-1为使用OpenGL ES 1.x渲染的都市赛车5场景，在该图中可以看出使用1.x渲染的3D场景不够真实，如路面和树木等。图3-2为使用OpenGL ES 2.0渲染的都市赛车6场景，在该图中可以看出使用OpenGL ES 2.0渲染出的路面及光影效果比OpenGL ES 1.x渲染出的更加真实。

通过都市赛车5及都市赛车6的效果图充分地说明了OpenGL ES 1.x与OpenGL ES 2.0渲染能力的差异。接下来将通过一款大家熟知的游戏——水果忍者，再次比较OpenGL ES 1.x与OpenGL ES 2.0渲染能力的差异。使用OpenGL ES 1.x与OpenGL ES 2.0渲染的水果忍者效果分别如图3-3和图3-4所示。



图3-3　OpenGL ES 1.x的渲染效果



图3-4　OpenGL ES 2.0的渲染效果

说明

由水果忍者的两幅渲染效果图比较，可以看出在视觉上OpenGL ES 2.0渲染的3D游戏场景比OpenGL ES 1.x系列渲染的3D游戏更加逼真。

接着再介绍两款使用OpenGL ES 2.0开发的3D游戏：激流快艇与刺客信条，分别如图3-5和 图3-6所示。这两款3D游戏均是大场景游戏，界面更加细腻、真实。



图3-5　激流快艇

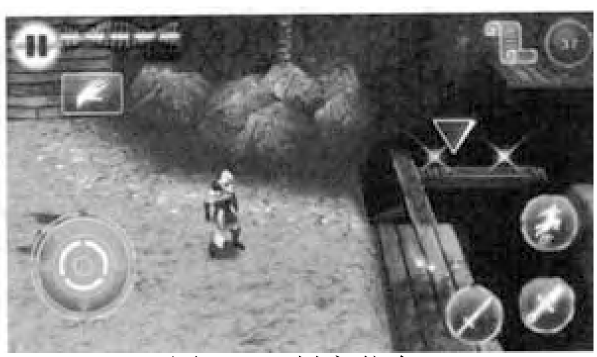


图3-6　刺客信条

说明

从图3-5激流赛艇以及图3-6刺客信条的效果图中可以看出，使用OpenGL ES 2.0渲染的游戏在细腻程度、真实感等方面有了质的变化，不再像使用OpenGL ES 1.x开发的3D游戏那样“原始感十足”。

介绍完上述激流赛艇以及刺客信条之后，不得不介绍鼎鼎大名的Gameloft旗下的两款著名3D游戏，地牢猎手和地牢守护者。这两款3D游戏一经发布便广受游戏爱好者的好评，其效果分别如图3-7和图3-8所示。



图3-7　地牢猎手



图3-8　地牢守护者

使用OpenGL ES 2.0不仅可以开发类似地牢猎手和地牢守护者等大场景的3D游戏，还可以开发较为真实的人物动作类型游戏，例如KONAMI的看家大作——实况足球2011，如图3-9所示。还有Indiagames出品的Bruce Lee Dragon Warrior（李小龙之龙战士），如图3-10所示。

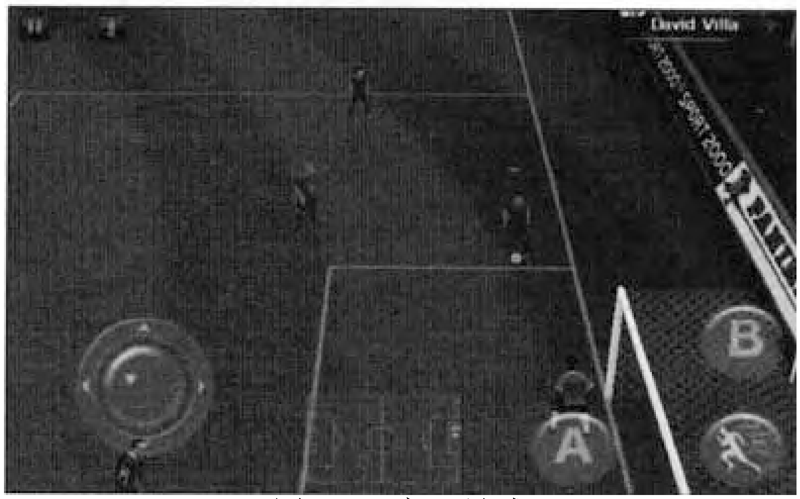


图3-9　实况足球



图3-10　李小龙之龙战士

前些年，想在智能手机上运行PC级的3D游戏，那简直是奢望。但是现在，那种原来不可能实现的愿望已经成为现实。随着手机硬件水平的不断提高、3D图形API的逐渐完善等多方面的因素，一些只能在PC上玩的大型游戏在Android手机上也可以安装并流畅运行了。

如EA的极品飞车，这款在PC上极其火爆的游戏，也已经成功移植到Android上了，效果如图3-11和图3-12所示。



图3-11　极品飞车13变速



图3-12　极品飞车14

说明

图3-11为EA开发的极品飞车13变速，图3-12为EA开发的极品飞车14热力追踪。这两款游戏均是以PC版的极品飞车13、14为原型，从而移植到Android手机中的。二者均使用的是OpenGL ES 2.0进行3D场景渲染，整体场景真实、细腻。

3.1.2　初识OpenGL ES 2.0应用程序

前一小节简单地介绍了OpenGL ES 2.0与OpenGL ES 1.x的不同之处，在本小节将通过一个旋转三角形的案例向读者介绍，如何使用OpenGL ES 2.0进行3D场景的开发。该案例运行效果分别如图3-13、图3-14和图3-15所示。

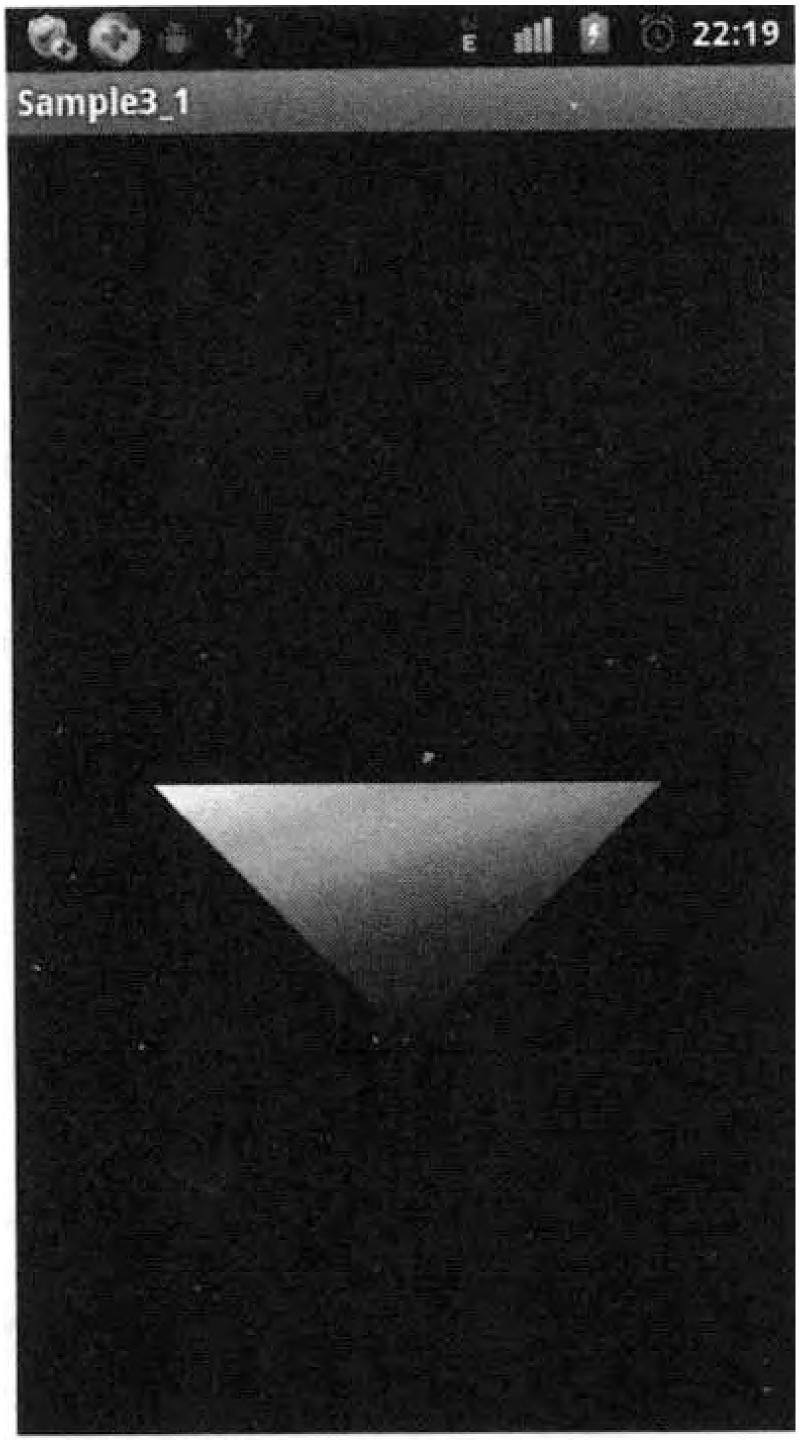


图3-13　运行效果图1

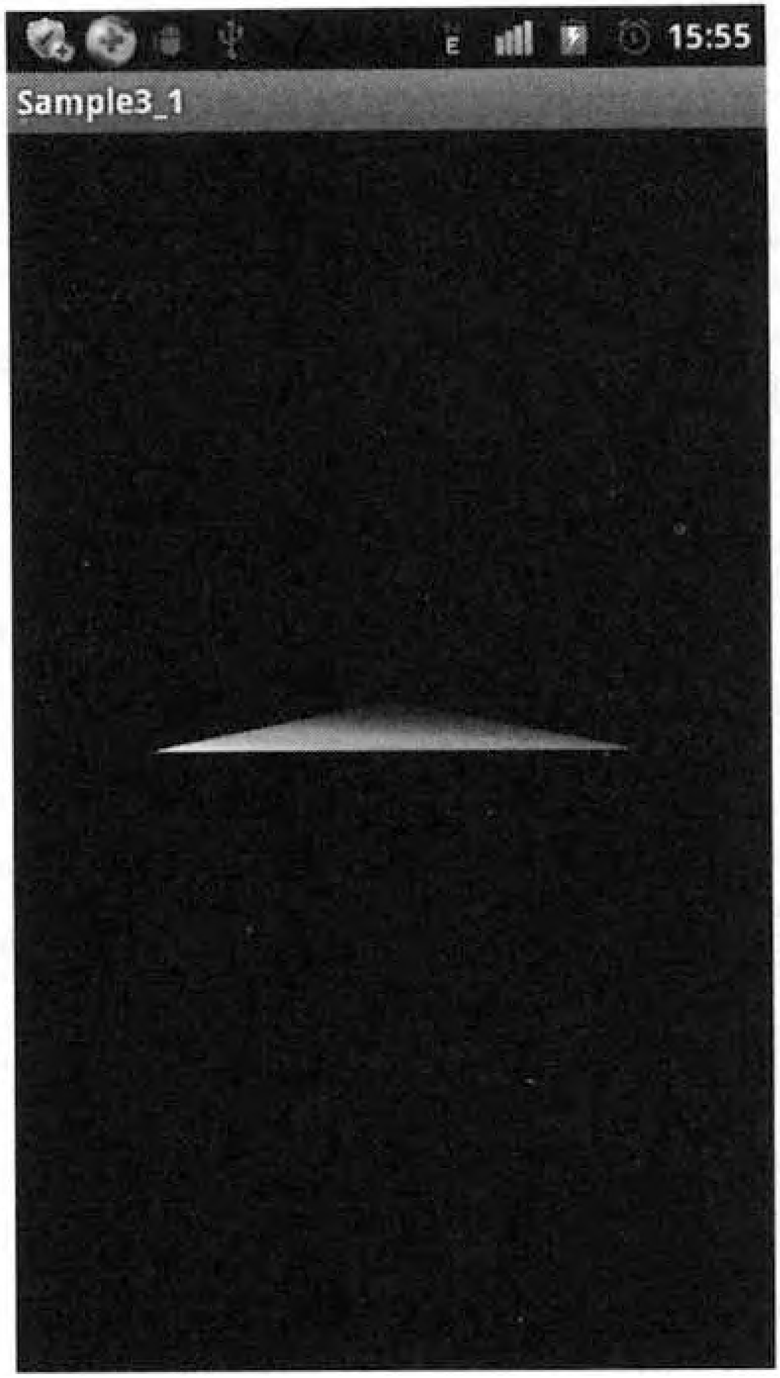


图3-14　运行效果图2

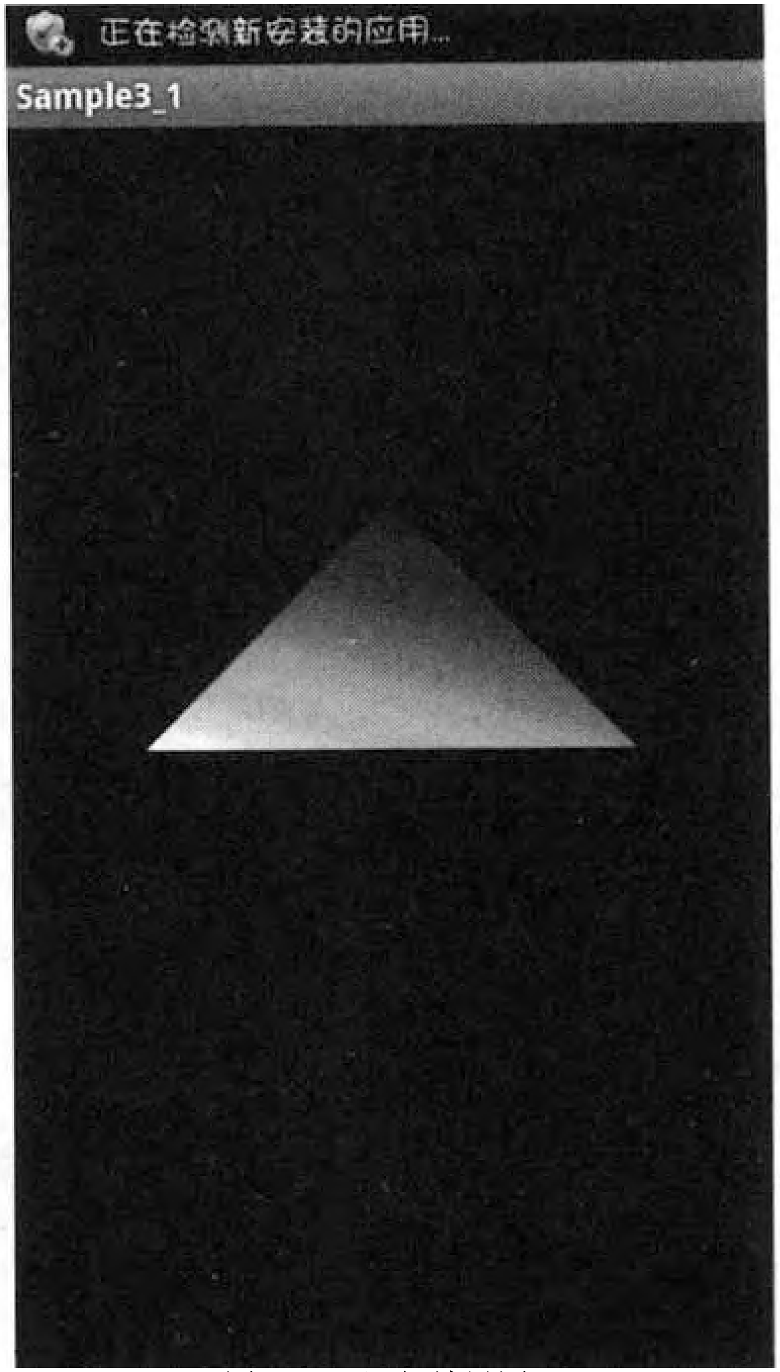


图3-15　运行效果图3

提示

请读者特别注意，基于OpenGL ES 2.0的3D应用程序目前还不能在模拟器上运行，必须使用配置了GPU（GPU要求支持OpenGL ES

2.0）的真机（Android版本要求最低为2.2）才可以。没有真机的读者想学习本书的内容可能需要购置或借用一部符合要求的真机了，关于GPU硬件的信息可以参考本章3.3节的相关内容。

说明

图3-13为3D空间中三角形初始状态的效果图、图3-14为3D空间中三角形绕*X*轴旋转大约100°的效果图、图3-15为绕*X*轴旋转大约180°的效果图。

前面已经介绍了本案例的运行效果，接下来将介绍本案例的具体开发步骤。

（1）在开发与本案例的功能直接相关的各种类之前，首先需要开发一个本书案例都需要用到的工具类——ShaderUtil，其功能为将着色器（Shader）脚本加载进显卡并编译。最先开发的是该类中从着色器sh脚本中加载着色器内容的loadFromAssetsFile方法和检查每一步操作是否有错误的checkGlError方法，具体代码如下。

代码位置：见随书光盘中源代码/第3章/Sample3_1/com/bn/Sample3_1目录下的ShaderUtil.java。

```
package com.bn.Sample3_1;                                          //声明包名
import java.io.ByteArrayOutputStream;                              //相关类的引入
......//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.util.Log;                                           //相关类的引入
public class ShaderUtil {                                          //加载顶点与片元着色器的类
  public static int loadShader(int shaderType, String source ){} //加载指定着色器的方法
  public static int createProgram(String vertexSource, String fragmentSource) { }
                                                                   //创建着色器程序的方法
  public static void checkGlError(String op) {                    //检查每一步操作是否有错误的方法
      int error;                                                   // error变量
          while ((error = GLES20.glGetError()) != GLES20.GL_NO_ERROR) {
             Log.e("ES20_ERROR", op + ": glError " + error);        //后台打印错误
             throw new RuntimeException(op + ": glError " + error); //抛出异常
      }}
      //从sh脚本中加载着色器内容的方法
      public static String loadFromAssetsFile(String fname,Resources r){
        String result=null;                              //声明一个String类型的字符串变量
        try{
```

```
18          InputStream in=r.getAssets().open(fname);    //从assets文件夹中读取信息
19             int ch=0;                                 //定义一个int型变量
20            ByteArrayOutputStream baos = new ByteArrayOutputStream();
21            while((ch=in.read())!=-1){
22                baos.write(ch);                        //将获取的信息写入输出流中
23            }
24            byte[] buff=baos.toByteArray();
25            baos.close();                              //关闭输出流
26            in.close();
27          result=new String(buff,"UTF-8");             //转化为UTF-8编码
28          result=result.replaceAll("\\r\\n","\n");
29       }
30       catch(Exception e){                             //捕获异常
31          e.printStackTrace();                         //打印异常
32       }
33       return result;                                  //返回结果
34   }}
```

- 第8-13行checkGlError方法的作用是在向GPU着色程序中加入顶点着色器或者片元着色器时，检查每一步操作是否有错误。这是由于在开发着色器脚本文件中的代码时，没有一个开发器实时地进行编译、查错，因此开发一个检查错误的方法尤为重要。
- 第14-34行的loadFromAssetsFile方法的作用为从项目根目录的assets文件夹下加载着色器代码脚本。其通过输入流将脚本信息读入，然后交给createProgram方法创建着色器程序。

（2）开发完加载着色器脚本内容的loadFromAssetsFile方法和检查错误的checkGlError方法后，下面开发的是加载着色器编码进GPU并进行编译的loadShader方法与创建着色器程序的createProgram方法，具体代码如下。

代码位置：见随书光盘中源代码/第3章/Sample3_1/com/bn/Sample3_1目录下的ShaderUtil.java。

```
public static int loadShader(                              //加载指定着色器的方法
        int shaderType,                                    //着色器的类型
        String source                                      //着色器的脚本字符串
) {
    int shader = GLES20.glCreateShader(shaderType);        //创建一个shader，并记录其id
    if (shader != 0) {                                     //若创建成功则加载着色器
      GLES20.glShaderSource(shader, source);               //加载着色器的源代码
      GLES20.glCompileShader(shader);                      //编译
      int[] compiled = new int[1];
         //获取Shader的编译情况
         GLES20.glGetShaderiv(shader, GLES20.GL_COMPILE_STATUS, compiled, 0);
         if (compiled[0] == 0) {        //若编译失败则显示错误日志并删除此shader
            Log.e("ES20_ERROR", "Could not compile shader " + shaderType + ":");
            Log.e("ES20_ERROR", GLES20.glGetShaderInfoLog(shader));
            GLES20.glDeleteShader(shader);
            shader = 0;
         }}
       return shader;
     }
    //创建着色器程序的方法
```

```
public static int createProgram(String vertexSource, String fragmentSource) {
                                                              //加载顶点着色器
    int vertexShader = loadShader(GLES20.GL_VERTEX_SHADER, vertexSource);
    if (vertexShader == 0) {
       return 0;
    }
                                                              //加载片元着色器
    int pixelShader = loadShader(GLES20.GL_FRAGMENT_SHADER, fragmentSource);
    if (pixelShader == 0) {
       return 0;
    }
    int program = GLES20.glCreateProgram();                   //创建程序
    if (program != 0) {              //若程序创建成功则向程序中加入顶点着色器与片元着色器
       GLES20.glAttachShader(program, vertexShader);   //向程序中加入顶点着色器
       checkGlError("glAttachShader");
       GLES20.glAttachShader(program, pixelShader);    //向程序中加入片元着色器
       checkGlError("glAttachShader");
       GLES20.glLinkProgram(program);                         //链接程序
       int[] linkStatus = new int[1];          //存放链接成功program状态值的数组
       GLES20.glGetProgramiv(program, GLES20.GL_LINK_STATUS, linkStatus, 0);
       if (linkStatus[0] != GLES20.GL_TRUE) {      //若链接失败则报错并删除程序
          Log.e("ES20_ERROR", "Could not link program: ");
          Log.e("ES20_ERROR", GLES20.glGetProgramInfoLog(program));
          GLES20.glDeleteProgram(program);          //删除程序
          program = 0;
       }}
    return program;                                      //返回结果
 }
```

• 第1-19行的加载指定着色器的方法中：第5行通过调用glCreateShader方法创建了一个着色器；如果着色器创建成功，第6-17行则加载着色器的源代码，并编译着色器，同时检测编译的情况。若编译成功则返回着色器id，反之则删除着色器并且打印错误信息。

• 第20-48行创建着色器程序的createProgram方法中：第23-31行通过调用loadShader方法，分别加载顶点着色器与片元着色器的源代码进GPU，并分别进行编译。第32-47行首先创建一个着色器程序，然后分别将相应的顶点与片元着色器添加到其中，最后将两个着色器链接为一个整体的着色器程序。

提示

到目前为止，读者可能还是对着色器一头雾水，不用担心，本章后面的内容将逐渐揭开着色器神秘的面纱。

（3）下面开始开发与本案例功能直接相关的类，首先需要开发的是本案例的主控制类Sample3_1Activity，该类继承自Activity，在程序开始时执行。该类的主要工作是创建MyTDView类的对象，然后调用setContentView方法跳转到相关界面，其代码如下。

代码位置：见随书光盘中源代码，第3章/Sample3_1/com/bn/Sample3_1目录下的Sample3_1Activity.java。

```
1 package com.bn.Sample3_1;                    //声明包名
2 import android.app.Activity;                 //相关类的引入
```

```
3 import android.content.pm.ActivityInfo;
4 import android.os.Bundle;                //相关类的引入
5 public class Sample3_1Activity extends Activity{
                                           //创建继承Activity的主控制类Sample3_1Activity
6     MyTDView mview;                      //声明MyTDView类的引用
7    @Override
8    public void onCreate(Bundle savedInstanceState){//继承Activity后重写的onCreate方法
9         super.onCreate(savedInstanceState);         //调用父类
10             setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
11             mview=new MyTDView(this);             //创建MyTDView类的对象
12             mview.requestFocus();                 //获取焦点
13             mview.setFocusableInTouchMode(true);  //设置为可触控
14             setContentView(mview);                //跳转到相关界面
15        }
16        @Override
17        public void onResume(){          //继承Activity后重写的onResume方法
18             super.onResume();
19             mview.onResume();           //通过MyTDView类的对象调用onResume方法
20        }
21        @Override
22        public void onPause(){           //继承Activity后重写的onPause方法
23             super.onPause();
24             mview.onPause();            //通过MyTDView类的对象调用onPause方法
25   }}
```

- 第6行为本类的成员变量，主要是声明MyTDView类的引用。
- 第7-15行为继承Activity后重写的onCreate方法，在该方法中主要是创建MyTDView类的对象，然后设置MyTDView获得焦点以及可触控，最后调用setContentView方法跳转到相关界面。

- 第16-20行为继承Activity后重写的onResume方法，在该方法中首先调用父类的onResume方法，然后调用MyTDView类对象的onResume方法。
- 第21-25行为继承Activity后重写的onPause方法，该方法中首先调用父类的onPause方法，然后调用MyTDView类对象的onPause方法。

（4）开发完本案例的主控制类Sample3_1Activity后，接下来需要开发的是本案例中用于创建三角形的类Triangle。该类的主要功能为初始化顶点数据、初始化着色器、设置相应的平移矩阵及旋转矩阵。首先介绍本类的基本框架，理解该框架有助于读者对本类整体结构的理解，其代码如下。

代码位置：见随书光盘中源代码/第3章/Sample3_1/com/bn/Sample3_1目录下的Triangle.java。

```
package com.bn.Sample3_1;                                    //声明包名
import java.nio.ByteBuffer;                                  //相关类的引入
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.opengl.Matrix;                                //相关类的引入
public class Triangle{
    public static float[] mProjMatrix = new float[16];       //4x4 投影矩阵
    public static float[] mVMatrix = new float[16];          //摄像机位置朝向的参数矩阵
    public static float[] mMVPMatrix;                        //总变换矩阵
    int mProgram;                                            //自定义渲染管线着色器程序 id
```

```
    int muMVPMatrixHandle;                              //总变换矩阵引用
    int maPositionHandle;                               //顶点位置属性引用
    int maColorHandle;                                  //顶点颜色属性引用
    String mVertexShader;                               //顶点着色器代码脚本
    String mFragmentShader;                             //片元着色器代码脚本
    static float[] mMMatrix = new float[16];
                                            //具体物体的 3D 变换矩阵，包括旋转、平移、缩放
    FloatBuffer  mVertexBuffer;                         //顶点坐标数据缓冲
    FloatBuffer  mColorBuffer;                          //顶点着色数据缓冲
    int vCount=0;                                       //顶点数量
    float xAngle=0;                                     //绕 x 轴旋转的角度
    public Triangle(MyTDView mv){                       //构造器
        initVertexData();                   //调用初始化顶点数据的 initVertexData 方法
        intShader(mv);                      //调用初始化着色器的 intShader 方法
    }
    public void initVertexData(){                       //自定义的初始化顶点数据的方法
      vCount=3;                                         //顶点数量为 3
      final float UNIT_SIZE=0.2f;                       //设置单位长度
      float vertices[]=new float[]{                     //顶点坐标数组
            -4*UNIT_SIZE,0,0, 0,-4*UNIT_SIZE,0, 4*UNIT_SIZE,0,0
      };
      ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
      vbb.order(ByteOrder.nativeOrder());               //设置字节顺序为本地操作系统顺序
      mVertexBuffer = vbb.asFloatBuffer();              //转换为浮点(Float)型缓冲
      mVertexBuffer.put(vertices);                      //在缓冲区内写入数据
      mVertexBuffer.position(0);                        //设置缓冲区起始位置
      float colors[]=new float[]{                       //顶点颜色数组
            1,1,1,0, 0,0,1,0, 0,1,0,0
      };
      ByteBuffer cbb = ByteBuffer.allocateDirect(colors.length*4);
      cbb.order(ByteOrder.nativeOrder());               //设置字节顺序为本地操作系统顺序
      mColorBuffer = cbb.asFloatBuffer();               //转换为浮点(Float)型缓冲
      mColorBuffer.put(colors);                         //在缓冲区内写入数据
      mColorBuffer.position(0);                         //设置缓冲区起始位置
    }
   ……//此处省略了初始化着色器以及绘制图形的方法，将在下面介绍
   public static float[]getFinalMatrix(float[] spec){   //产生最终变换矩阵的方法
        mMVPMatrix=new float[16];                       //初始化总变换矩阵
        Matrix.multiplyMM(mMVPMatrix, 0, mVMatrix, 0, spec, 0);
        Matrix.multiplyMM(mMVPMatrix, 0, mProjMatrix, 0, mMVPMatrix, 0);
        return mMVPMatrix;                              //返回总变换矩阵
}}
```

- 第6-19行为本类所需成员变量的声明，主要是声明相关矩阵的引用、自定义渲染管线着色器程序的id、顶点位置和颜色属性的引用、顶点着色器以及片元着色器代码脚本字符串、顶点坐标和顶点着色数据缓冲、顶点的数量以及绕x轴旋转的角度。
- 第20-23行为本类的构造器，在该构造器中主要是调用initVertexData方法初始化顶点相关 的数据，并且调用intShader方法初始化着色器。
- 第24-43行为初始化顶点数据的initVertexData方法。该方法中需要指定顶点的坐标数据以及顶点的颜色数据，将数据写入到对应的缓冲区中，并设置缓冲区的起始位置。
- 第45-50行为通过物体的3D变换矩阵、摄像机参数矩阵、投影矩阵计算产生最终总变换矩阵的方法。关于各种变换矩阵的问题读者不用担心，本书后面将有专门的章节详细介绍。

（5）开发完Triangle类的基本框架后，接下来将开发初始化着色器的方法initShader，以及绘制三角形的方法drawSelf,，将下列代码插入到上述Triangle类框架的第44行。



代码位置：见随书光盘中源代码/第3章/Sample3_1/com/bn/Sample3_1目录下的Triangle.java。

```
1 public void intShader(MyTDView mv){                           //创建并初始化着色器的方法
2     mVertexShader=ShaderUtil.loadFromAssetsFile("vertex.sh", mv.getResources());
3     mFragmentShader=ShaderUtil.loadFromAssetsFile("frag.sh", mv.getResources());
4     mProgram = ShaderUtil.createProgram(mVertexShader, mFragmentShader);
5     maPositionHandle = GLES20.glGetAttribLocation(mProgram, "aPosition");
6     maColorHandle= GLES20.glGetAttribLocation(mProgram, "aColor");
7     muMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
8 }
9 public void drawSelf(){                                        //自定义的绘制三角形的方法
10        GLES20.glUseProgram(mProgram);
11        Matrix.setRotateM(mMMatrix,0,0,0,1,0);                 //初始化变换矩阵
12        Matrix.translateM(mMMatrix,0,0,0,1);                   //设置沿z轴正向位移
13        Matrix.rotateM(mMMatrix,0,xAngle,1,0,0);               //设置绕x轴旋转
14        GLES20.glUniformMatrix4fv(muMVPMatrixHandle,1,false,Triangle. GetFinalMatrix
          (mMMatrix), 0);
15        GLES20.glVertexAttribPointer(                          //将顶点位置数据传送进渲染管线
16        maPositionHandle,3,GLES20.GL_FLOAT,false,3*4,mVertexBuffer
17        );
18        GLES20.glVertexAttribPointer(                          //将顶点颜色数据传送进渲染管线
19            maColorHandle,4,GLES20.GL_FLOAT,false,4*4,mColorBuffer
20        );
21        GLES20.glEnableVertexAttribArray(maPositionHandle);  //启用顶点位置数据
22        GLES20.glEnableVertexAttribArray(maColorHandle);     //启用顶点着色数据
23        GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount); //执行绘制
24    }
```

- 第1-8行为初始化着色器的方法。在该方法中首先需要加载相应的着色器脚本，然后创建自定义的渲染管线着色器程序，并保留程

序id于mProgram中。接着通过GLES20类调用相应的方法获取着色器程序中顶点坐标数据的引用、顶点颜色数据的引用以及总变换矩阵的引用。

- 第10行通过GLES20类调用glUseProgram方法给出着色器程序id指定使用的着色器程序。
- 第11-14行功能为初始化变换矩阵，设置沿z轴正方向的位移值以及绕*x* 轴旋转的角度值。
- 第15-20行为通过调用GLES20类的glVertexAttribPointer方法，将顶点坐标数据以及顶点颜色数据传送进渲染管线，以备渲染时在顶点着色器中使用。
- 第21-23行为通过调用GLES20类的glEnableVertexAttribArray方法启用顶点位置数据以及启用顶点颜色数据，并通过调用GLES20类的glDrawArrays方法绘制三角形。

提示

本类中用到的有关着色器的知识、平移以及旋转变换的相关知识，将在后面的章节中逐步进行详细的介绍，在本案例中由于知识不全，不对这些进行详细解释。

（6）开发完本案例的主控制类Sample3_1Activity以及三角形类Triangle后，接下来将开发本案例中用于显示3D场景的类MyTDView。该类继承自GLSurfaceView，并且在该类中通过内部类的形式创建了渲染器，其代码如下。

代码位置：见随书光盘中源代码/第3章/Sample3_1/com/bn/Sample3_1目录下的MyTDView.java。

```
package com.bn.Sample3_1;                              //声明包名
import javax.microedition.khronos.egl.EGLConfig;//相关类的引入
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.view.MotionEvent;                       //相关类的引入
public class MyTDView extends GLSurfaceView{     //创建继承 GLSurfaceView 的 MyTDView 类
    //每次三角形旋转的角度
    final float ANGLE_SPAN = 0.375f;
    RotateThread rthread;                               //自定义线程类 RotateThread 的引用
    SceneRenderer mRenderer;                            //自定义渲染器的引用
        public MyTDView(Context context){               //构造器
            super(context);
            this.setEGLContextClientVersion(2);         //使用 OpenGL ES 2.0 需设置该值为 2
            mRenderer=new SceneRenderer();              //创建 SceneRenderer 类的对象
            this.setRenderer(mRenderer);                //设置渲染器
            this.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
        }
        private class SceneRenderer implements GLSurfaceView.Renderer{
            Triangle tle;                               //声明 Triangle 类的引用
            public void onDrawFrame(GL10 gl){           //重写的 onDrawFrame 方法
            GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT | GLES20.GL_COLOR_BUFFER_BIT);
             tle.drawSelf();            //通过 Triangle 的对象调用 drawSelf 绘制三角形
            }
            public void onSurfaceChanged(GL10 gl, int width, int height){
                GLES20.glViewport(0, 0, width, height); //设置视口
                float ratio = (float) width / height;  //计算屏幕的宽度和高度比例
                Matrix.frustumM(Triangle.mProjMatrix, 0, -ratio, ratio, -1, 1, 1, 10);
                                                        //设置透视投影
                Matrix.setLookAtM(Triangle.mVMatrix,0,0,0,3,0f,0f,0f,0f,1.0f,0.0f);
                                                        //设置摄像机
            }
            public void onSurfaceCreated(GL10 gl, EGLConfig config){
                GLES20.glClearColor(0,0,0,1.0f);        //设置屏幕背景色
                tle=new Triangle(MyTDView.this);        //创建 Triangle 类的对象
                GLES20.glEnable(GLES20.GL_DEPTH_TEST);
                rthread=new RotateThread();             //创建 RotateThread 类的对象
                rthread.start();                        //开启线程
        } }
        public class RotateThread extends Thread{       //自定义的内部类线程
```

```
37            public boolean flag=true;                //设置循环标志位
38            @Override
39            public void run(){                       //重写的run方法
40                while(flag){                         //while循环
41                    mRenderer.tle.xAngle=mRenderer.tle.xAngle+ ANGLE_SPAN;
42                    try{
43                        Thread.sleep(20);            //线程休眠20ms
44                    }catch(Exception e){             //捕获并打印异常信息
45                        e.printStackTrace();
46    }}}}}
```

- 第6-9行为本类的成员变量，主要有每次旋转的角度、旋转线程类RotateThread对象的引用以及自定义渲染器SceneRenderer的引用。
- 第10-16行为本类的构造器，在该构造器中创建了SceneRenderer类的对象，设置了渲染器，并且设置渲染模式为主动渲染。
- 第17-22行为实现GLSurfaceView.Renderer接口后重写的onDrawFrame方法，在该方法中首先需要清除深度缓存和颜色缓存，然后通过Triangle类的对象调用drawSelf方法绘制三角形。
- 第23-28行为实现GLSurfaceView.Renderer接口后重写的onSurfaceChanged方法，在该方法中设置了视口、透视投影以及摄像机的位置。
- 第29-35行为实现GLSurfaceView.Renderer接口后重写的onSurfaceCreated方法，在该方法中设置了屏幕的背景颜色、创建了Triangle类的对象、开启了深度检测、创建了RotateThread类的对象并启动了该线程。
- 第36-46行为自定义的内部类线程，在该线程中通过while循环不断更改Triangle类中xAngle的值，使得三角形以一定的角速度绕x轴转动。

提示

本案例采用的用到变换矩阵时临时添加变换矩阵的方式并不方便，在本书后面的章节中，这些矩阵将会被集合到一个MatrixState工具类当中，使得开发人员在应用时更加方便。

（7）完成了Java代码的开发后，接着就可以用着色器语言开发着色器了，着色器的代码可以存储在后缀名为“.sh”文件中，这些文件存放到项目的assets目录下。首先开发的是顶点着色器，主要作用为执行顶点变换、纹理坐标变换等顶点的相关操作，具体代码如下。

代码位置：见随书光盘中源代码/第3章/Sample3_1/com/bn/Sample3_1/assets目录下的vertex.sh文件。

```
uniform mat4 uMVPMatrix;                          //总变换矩阵
attribute vec3 aPosition;                         //顶点位置
attribute vec4 aColor;                            //顶点颜色
varying vec4 vColor;                              //用于传递给片元着色器的易变变量
void main() {
   gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点的位置
   vColor = aColor;                               //将接收的顶点颜色传递给片元着色器
}
```

说明

关于着色语言以及顶点着色器的作用将会在后面的章节中逐步详细介绍，这里对于顶点着色器中的代码不进行详细的讲解，读者也只要简单了解即可。

（8）完成了顶点着色器代码的开发后，下面将开发的是片元着色器的代码，其主要作用为执行纹理的访问、颜色的汇总以及雾效等操

作，具体代码如下。

代码位置：见随书光盘中源代码/第3章/Sample3_1/com/bn/Sample3_1/assets目录下的frag.sh文件。

```
precision mediump float;
varying  vec4 vColor;                    //接收从顶点着色器过来的易变变量
void main() {                            //main 方法
   gl_FragColor = vColor;                //给此片元赋颜色值
}
```

> **说明**
>
> 关于着色语言以及片元着色器的作用将会在后面的章节中逐步详细介绍，这里对于片元着色器中的代码不进行详细的讲解，读者也只要简单了解即可。

3.2 着色器与渲染管线

前面一节已经通过一个三角形的案例为读者简单介绍了如何使用OpenGL ES 2.0进行3D场景的开发，相信读者对OpenGL ES 2.0应用的开发已经有了初步的了解。但要想真正地掌握OpenGL ES 2.0，则首先要掌握着色器和渲染管线相关知识，本节将向读者介绍这方面的内容。

3.2.1 OpenGL ES 1.x的渲染管线

学习OpenGL ES 2.0的渲染管线之前，应该首先了解一下OpenGL ES 1.x的渲染管线，这对于进一步深入了解OpenGL ES 2.0的渲染管线是很有裨益的，本小节将对此内容进行介绍。

渲染管线有时也称之为渲染流水线，一般是由显示芯片（GPU）内部处理图形信号的并行处理单元组成。这些并行处理单元两两之间是相互独立的，在不同型号的硬件上独立处理单元的数量也有很大的差异。一般越高端型号的硬件，其中独立处理单元的数量也就越多。

提示

在有些没有GPU硬件的设备上也有采用软件模拟实现管线中的各个处理单元的情况，一般多见于廉价的低端设备。

与普通应用程序通过CPU串行执行不同，将渲染工作通过渲染管线中多个相互独立的处理单元进行并行处理后，渲染效率可以得到极大的提升。

从另一个角度看，OpenGL ES中的渲染管线实质上指的是一系列绘制过程。这些过程输入的是待渲染3D物体的相关描述信息数据，经过渲染管线，输出的是一帧想要的图像。OpenGL ES 1.x的渲染管线如图3-16所示。

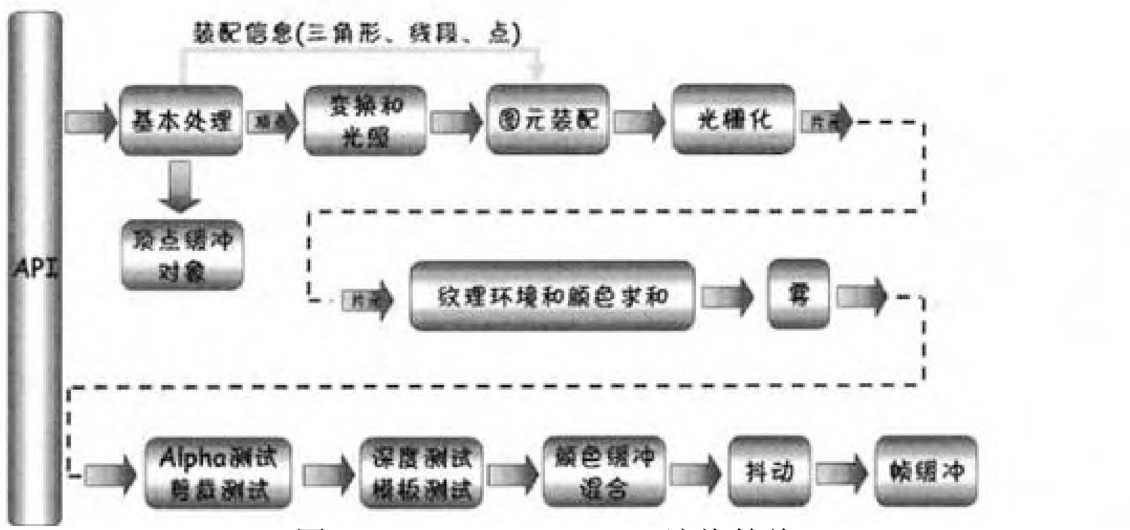


图3-16　OpenGL ES 1.x渲染管线

1．基本处理

该阶段设定3D空间中物体的顶点坐标、顶点对应的颜色、顶点的纹理坐标等属性，并且指定 绘制方式，如：点绘制、线段绘制或者三

角形绘制等。

2．顶点缓冲对象

这部分功能在应用程序中是可选的，对于某些在整个场景中顶点的基本数据不变的情况。可以在初始化阶段将顶点数据经过基本处理后送入顶点缓冲对象，在绘制每一帧想要的图像时就省去了顶点数据IO的麻烦，直接从顶点缓冲对象中获取顶点数据即可。相比于每次绘制时单独将顶点数据送入GPU的方式，可以在一定程度上节省GPU的IO带宽，提高渲染效率。

3．变换和光照

该阶段的工作主要是进行顶点变换以及根据程序中设置的光照属性对顶点进行光照计算。顶点变换的任务是对3D物体的各顶点进行平移、旋转或者缩放等操作。光照计算的任务是根据程序送入的光源位置、性质、各通道强度、物体的材质等，同时再根据一定的光照数学模型计算各顶点的光照情况。

4．图元装配

这个阶段主要有两个任务，一个是图元组装，另一个是图元处理。所谓图元组装是指顶点数据根据设置的绘制方式被结合成完整的图元。例如，点绘制方式仅需要一个单独的顶点，此方式下每个顶点为一个图元；线段绘制方式则需要两个顶点，此方式下每两个顶点构成一个图元；三角形绘制方式下需要3个顶点构成一个图元。

图元处理最重要的工作是剪裁，其任务是消除位于半空间（half-space）之外的部分几何图元，这个半空间是由一个剪裁平面所定义的。例如，点剪裁就是简单地接受或者拒绝顶点，线段或多边形剪裁可能需要增加额外的顶点，具体取决于直线或者多边形与剪裁平面之间的位置关系，如图3-17所示。

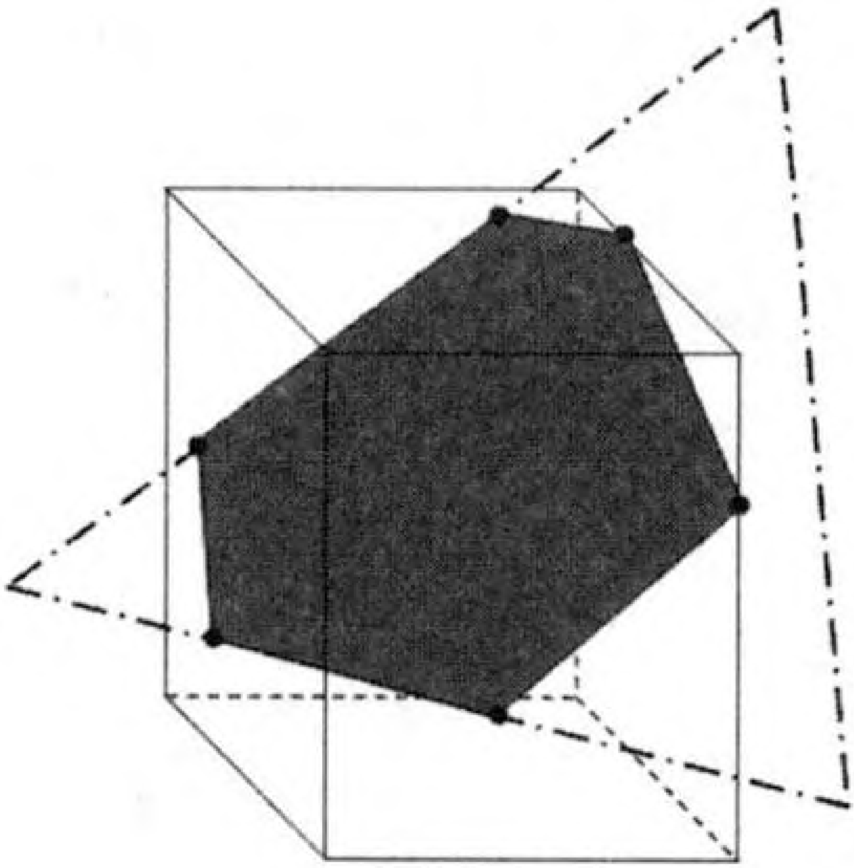


图3-17　剪裁三角形3个顶点生成6个新的顶点

说明

图3-17给出了一个三角形图元（图中为点划线绘制）被4个剪裁平面剪裁的情况。4个剪裁平面分别为：上面、左侧面、右侧面、后面。

之所以要进行剪裁是因为随着观察位置、角度的不同，并不总能看到（这里可以简单地理解为显示到设备屏幕上）特定3D物体某个图元的全部。例如，当观察一个正四面体并离某个三角形面很近时，可能只能看到此面的一部分，这时在屏幕上显示的就不再是三角形了，而是经过裁剪后形成的多边形，如图3-18所示。

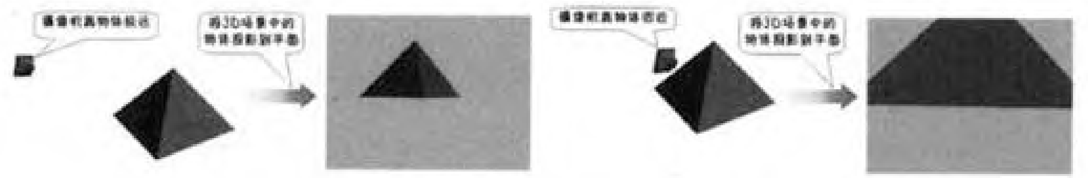


图3-18　从不同角度、距离观察正四面体

剪裁时，若图元完全位于视景体以及自定义剪裁平面的内部，则将图元传递到后面的步骤进行处理；如果其完全位于视景体或者自定义剪裁平面的外部，则丢弃该图元；如果其有一部分位于内部，另一部分位于外部，则需要剪裁该图元。

提示

关于视景体剪裁的问题会在第5章介绍投影的部分进行详细的介绍，这里读者简单了解即可。

5．光栅化

虽然虚拟3D世界中的几何信息是三维的，但由于目前用于显示的设备都是二维的。因此在真正执行光栅化工作之前，首先需要将虚拟3D世界中的物体投影到视平面上。需要注意的是，由于观察位置的不同，同一个3D场景中的物体投影到视平面可能会产生不同的效果，如图3-19所示。

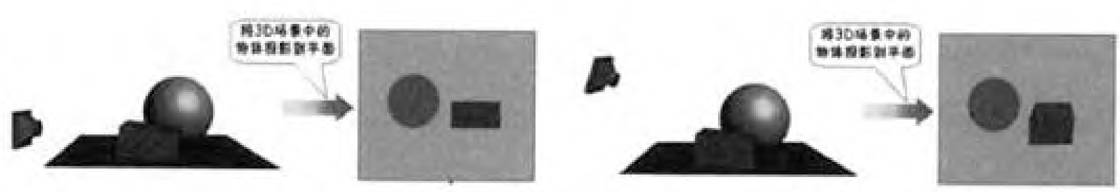


图3-19　光栅化阶段投影到视口

另外，由于虚拟3D世界当中物体的几何信息一般采用连续的数学量来表示，因此投影的平面结果也是用连续数学量表示的。但目前的显示设备屏幕都是离散化的（由一个一个的像素组成），因此还需要

将投影的结果离散化。将其分解为一个一个离散化的小单元，这些小单元一般称之为片元，具体效果如图3-20所示。

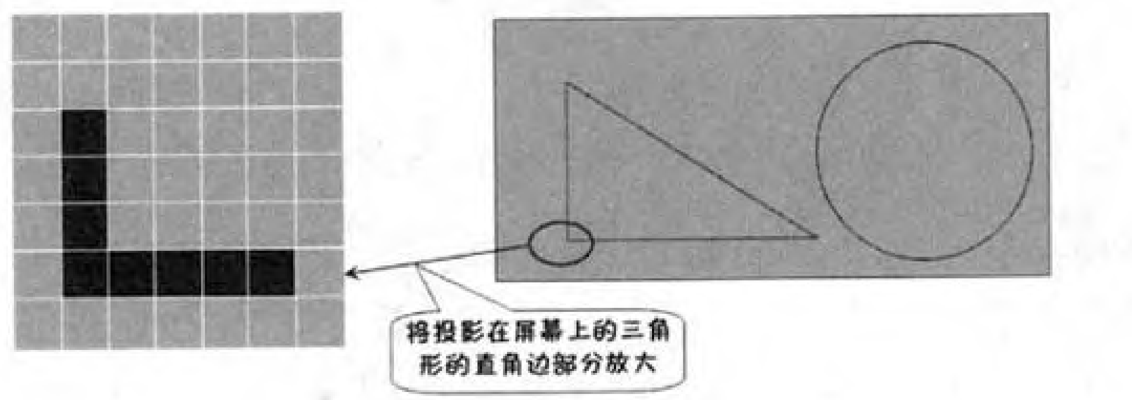


图3-20　投影后图元离散化

其实每个片元都对应于帧缓冲中的一个像素，之所以不直接称之为像素是因为3D空间中的物体是可以相互遮挡的。而一个3D场景最终显示到屏幕上虽然是一个整体，但每个3D物体的每个图元是独立处理的。这就可能出现这样的情况，系统先处理的是位于离观察点较远的图元，其光栅化成为了一组片元，暂时送入帧缓冲的对应位置。

但后面继续处理离观察点较近的图元时也光栅化出了一组片元，两组片元中有对应到帧缓冲中同一个位置的，这时距离近的片元将覆盖距离远的片元（如何覆盖的检测是在深度检测阶段完成）。因此某片元就不一定能成为最终屏幕上的像素，称之为像素就不准确了，可以将其理解为候选像素。

提示

每个片元包含其对应的顶点坐标、顶点颜色、顶点纹理坐标以及顶点的深度等信息，这些信息是系统根据投影前此片元对应的3D空间中的位置及与此片元相关的图元的各顶点信息进行插值计算而生成的。

6．纹理环境和颜色求和

该阶段主要有两个任务，具体如下所列。

- 一个是纹理采样任务，其主要是根据当前需处理片元的纹理坐标及采用的纹理id对相应的纹理图进行纹理采样，获取采样值。

> **提示**
>
> 所谓纹理采样值就是从纹理图中某个纹理坐标位置获取的一个颜色值，这方面的细节问题在后面专门介绍纹理的章节将详细介绍，这里读者简单了解即可。

- 另一个是颜色求和，主要是执行颜色的变化，其根据纹理采样及光照计算等的结果综合生成需处理片元的颜色。

7．雾

该阶段主要是根据程序中设置的雾的相关参数，如：颜色、浓度、范围等，以及某种雾的数学 模型来计算当前处理的片元受雾影响后的颜色。

8．Alpha测试

如果程序中启用了Alpha测试，OpenGL ES会检查每个片元的Alpha值，只有Alpha值符合测试条件的片元才会送入下一阶段，不满足条件的片元则被丢弃。

9．剪裁测试

如果程序中启用了剪裁测试，OpenGL ES会检查每个片元在帧缓冲中对应的位置，若对应位置在剪裁窗口中则将此片元送入下一阶段，否则丢弃此片元。

10．深度测试和模板测试

- 深度测试是指将输入片元的深度值与帧缓冲区中存储的对应位置片元的深度值进行比较，若输入片元的深度值小则将输入片元送入

下一阶段准备覆盖帧缓冲中的原片元或与帧缓冲中的原片元混合，否则丢弃输入片元。

- 模板测试的主要功能为将绘制区域限定在一定的范围内，一般用在湖面倒影、镜像等场合，后面的章节会详细介绍。

11．颜色缓冲混合

若程序中开启了Alpha混合，则根据混合因子将上一阶段送来的片元与帧缓冲中对应位置的片元进行Alpha混合；否则送入的片元将覆盖帧缓冲中对应位置的片元。

12．抖动

抖动是一种简单的操作，其允许只使用少量的颜色模拟出更宽的颜色显示范围，从而使颜色视觉效果更加地丰富。例如，可以使用白色以及黑色模拟出一种过渡的灰色。

但使用抖动也是有其固有缺点的，那就是会损失一部分分辨率，因此对于现在主流的原生颜色就很丰富的显示设备一般是不需要启用抖动的。

> **提示**
>
> 当下的一些系统中虽然在API方面支持开启抖动，但这仅仅是为了API的兼容，其可能根本不会去进行事实上的抖动操作。

13．帧缓冲

OpenGL ES中的物体绘制并不是直接在屏幕上进行的，而是预先在帧缓冲区中进行绘制，每绘制完一帧再将绘制的结果交换到屏幕上。因此，在每次绘制新的一帧时都需要清除缓冲区中的相关数据，否则有可能产生不正确的绘制效果。

同时需要了解的是为了应对不同方面的需要，帧缓冲是由一套组件组成的，主要包括颜色缓冲、深度缓冲以及模板缓冲，各组件的具体用途如下所列。

- 颜色缓冲用于存储每个片元的颜色值，每个颜色值包括RGBA（红、绿、蓝、透明度）4个 色彩通道，应用程序运行时在屏幕上看到的就是颜色缓冲中的内容。
- 深度缓冲用来存储每个片元的深度值，所谓深度值是指以特定的内部格式表示的从片元处到观察点（摄像机）的距离。在启用深度测试的情况下，新片元想进入帧缓冲时需要将自己的深度值与帧缓冲中对应位置片元的深度值进行比较，若结果为小于才有可能进入缓冲，否则被丢弃。
- 模板缓冲用来存储每个片元的模板值，供模板测试使用。模板测试是几种测试中最为灵活和复杂的一种，后面将由专门的章节进行介绍。

提示

本小节只是对渲染管线中的每一个模块进行了简单的介绍，更为具体的情况会在本书中的后继章节进行更为详细的讨论，这里读者只要在概念上有个整体的把握即可。

3.2.2　OpenGL ES 2.0的渲染管线

OpenGL ES 1.x只是对开发人员开放了其中的一些API接口，但在整个渲染管线的运行过程中开发人员是不能直接干预的。因此，虽然OpenGL ES 1.x的渲染管线功能已经很强大，但是其留给编程人员发挥的空间不大，很多特效难以开发，而OpenGL ES 2.0则为编程人员提供了更多的发挥空间。

总的来说，OpenGL ES 2.0与OpenGL ES 1.x渲染管线还是存在着许多的相似性，只是OpenGL ES 2.0管线中有一些部分已经改变。在这些改变的部分中，编程人员可以插入自己的一些特殊操作，方便高效地完成OpenGL ES 1.x难以完成的任务，OpenGL ES 2.0可编程渲染管线如图3-21所示。

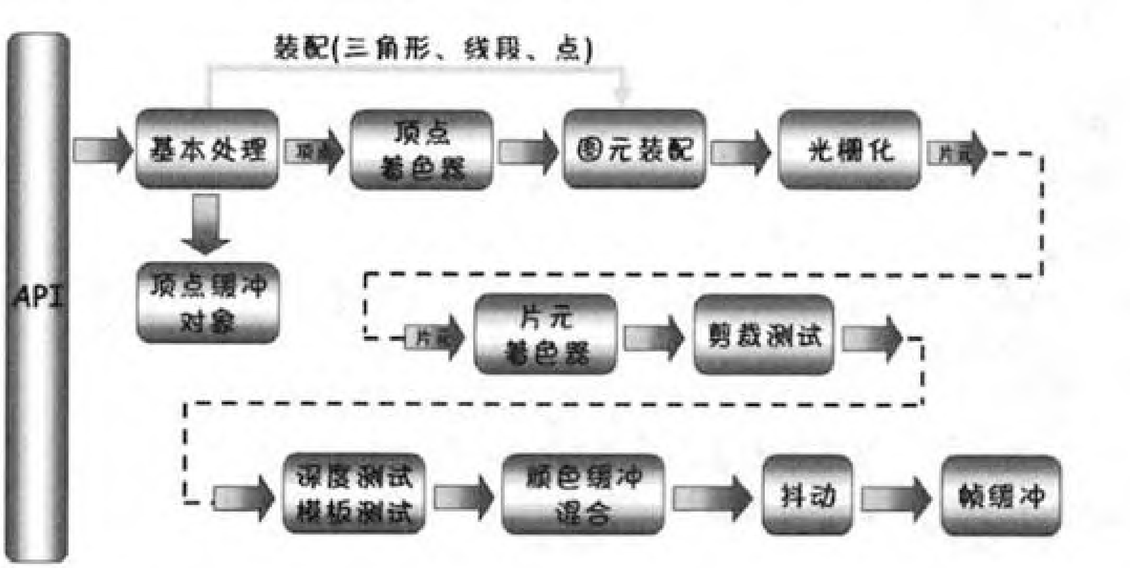


图3-21 OpenGL ES 2.0可编程渲染管线

从图3-21中可以看出，虽然有很多相似之处，但OpenGL ES 2.0与1.x的渲染管线还是有不少区别的，具体如下所列。

- OpenGL ES 2.0中“顶点着色器”取代了OpenGL ES 1.x渲染管线中的“变换和光照”阶段，这使得开发3D场景时对顶点的变换、法向量的计算、纹理坐标的变换、光照与材质的应用等均由开发者使用着色器代码完成，灵活性大大增强，可以轻松实现很多OpenGL ES 1.x难以完成的顶点处理任务。
- OpenGL ES 2.0中的“片元着色器”取代了OpenGL ES 1.x渲染管线中的“纹理环境和颜色求和”、“雾”以及“Alpha测试”等阶段，这使得纹理处理、颜色求和以及雾效果均由开发者自行开发，大大增强了程序对片元的处理能力，可以轻松实现很多OpenGL ES 1.x难以完成的片元处理任务。

提示

由于在前一小节OpenGL ES 1.x的渲染管线中已经详细介绍了各个阶段的作用，所以对于OpenGL ES 2.0渲染管线与OpenGL ES 1.x渲染管线中相同的部分这里就不再赘述，有需要的读者请参考前一小节中的相关内容。

从前面的内容中可以看出，OpenGL ES 2.0与OpenGL ES 1.x渲染管线的最大区别就是采用可编程顶点着色器与片元着色器替代了原有固定管线的很多功能，下面就对这两种重要的着色器进行介绍。

1．顶点着色器

顶点着色器是一个可编程的处理单元，功能为执行顶点的变换、光照、材质的应用与计算等顶点的相关操作，其每顶点执行一次。其工作过程为首先将原始的顶点几何信息及其他属性传送到顶点着色器中，经过自己开发的顶点着色器处理后产生纹理坐标、颜色、点位置等后继流程需要的各项顶点属性信息，然后将其传递给图元装配阶段。

顶点着色器替代了原有固定管线的顶点变换、光照计算，开发人员可以根据自己的需求自行开发顶点变换、光照等功能，大大增加了程序的灵活性。但凡事有利皆有弊，增加灵活性的同时也增加了开发的难度。由于这个原因，对初学者而言可能会感觉OpenGL ES 2.0比OpenGL ES 1.x难很多，不易上手。顶点着色器的工作原理如图3-22所示。

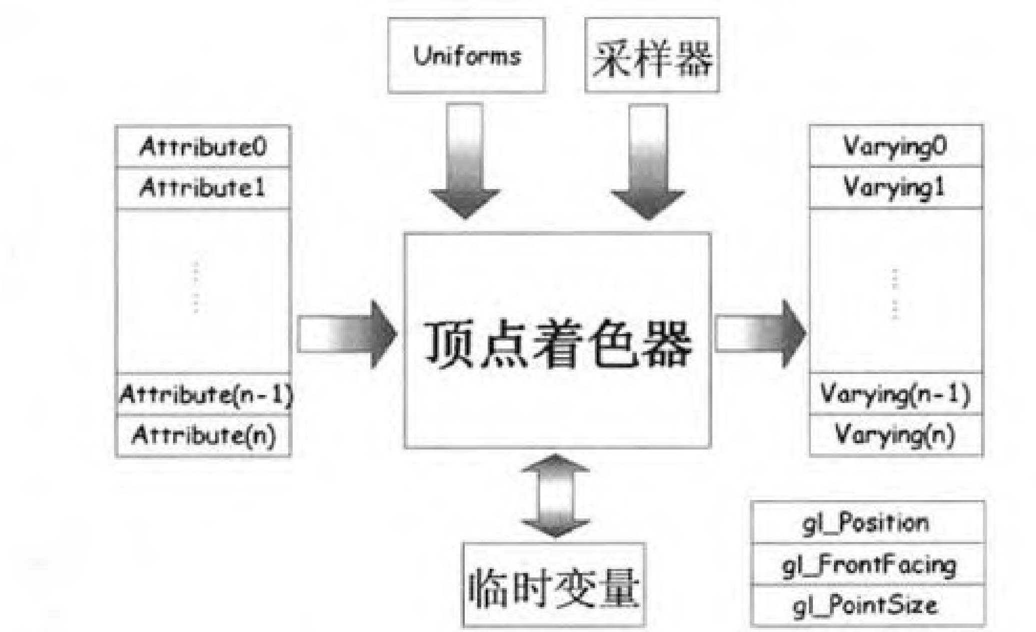


图3-22　OpenGL ES 2.0顶点着色器工作原理

- 顶点着色器的输入主要为待处理顶点相应的attribute（属性）变量、uniform（全局）变量、采样器以及临时变量，输出主要为

经过顶点着色器后生成的varying（易变）变量及一些内建输出变量。

- attribute变量指的是3D物体中每个顶点各自不同的信息所属的变量，一般顶点的位置、颜色、法向量等每个顶点各自不同的信息都是以attribute变量的方式传入顶点着色器的。例如，前一节“初识OpenGL ES 2.0应用程序”中顶点着色器里的aPosition（顶点位置）和aColor（顶点颜色）变量等。
- uniform变量指的是对于同一组顶点组成的单个3D物体中所有顶点都相同的量，一般为场景中当前的光源位置、当前的摄像机位置、投影系列矩阵等。例如，前一节“初识OpenGL ES 2.0应用程序”案例中顶点着色器里的uMVPMatrix（总变换矩阵）变量等。
- varying变量（易变变量）是从顶点着色器计算产生并传递到片元着色器的数据变量。顶点着色器可以使用易变变量来传递需要插值到片元的颜色、法向量、纹理坐标等任意值。例如，前一节OpenGL ES 2.0应用程序中由顶点着色器传入片元着色器中的aaColor变量。
- 内建输出变量gl_Position、gl_FrontFacing和gl_PointSize等。gl_Position是经过变换矩阵变换、投影后的顶点的最终位置，gl_FrontFacing指的是片元所在面的朝向，gl_PointSize指的是点的大小。

易变变量在顶点着色器赋值后并不是直接将赋的值送入到后继的片元着色器中，而是在光栅化阶段由管线根据片元所属图元各个顶点对应的顶点着色器对此易变变量的赋值情况及片元与各顶点的位置关系插值产生，图3-23所示说明了这个问题。

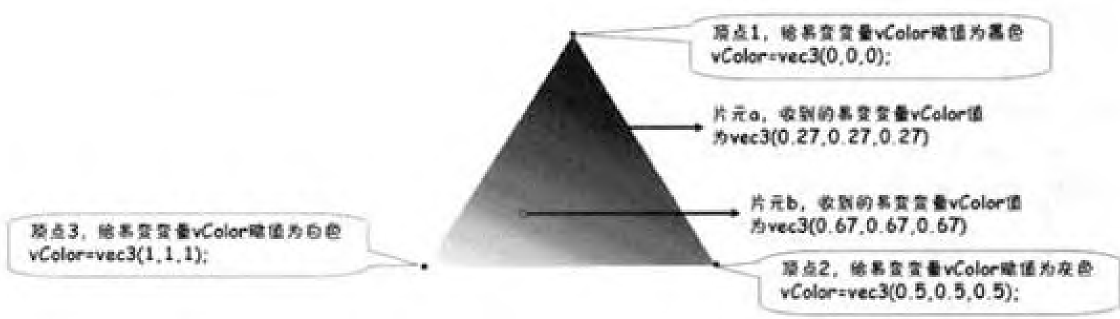


图3-23　易变变量的工作原理

提示

有一些数学知识的读者可能会想到一个问题，对每个片元进行一次插值计算将会非常耗费时间，严重影响性能。幸运的是，

OpenGL ES 2.0的设计者也考虑到了这个问题，这些插值操作都是由GPU中的专用硬件实现的，因此速度很快，不影响性能。

2. 片元着色器

片元着色器是用于处理片元值及其相关数据的可编程单元，其可以执行纹理的采样、颜色的汇总、计算雾颜色等操作，每片元执行一次。片元着色器主要功能为通过重复执行（每片元一次），将3D物体中的图元光栅化后产生的每个片元的颜色等属性计算出来送入后继阶段，如剪裁测试、深度测试及模板测试等。

从渲染管线的图中可以看出，可编程片元着色器替代了纹理、颜色求和、雾以及Alpha测试等阶段。与顶点着色器类似，被其替代的功能系统将不再提供，需要完全由开发人员用着色器语 言编程完成。这在提高了灵活性的同时也增加了开发的难度，尤其是对于初学者，其基本工作原理如图3-24所示。



图3-24　炮弹开始飞出

- Varying0～n指的是从顶点着色器传递到片元着色器的易变数据变量，如前面所介绍，由系统在顶点着色器后的光栅化阶段自动插值产生。其个数是不一定的，取决于具体的需要。
- gl_FragColor指的是计算后此片元的颜色。一般在片元着色器的最后都需要对gl_FragColor进行赋值。例如，上一节旋转三角形案例的片元着色器中就有“gl_FragColor = aaColor”语句。

提示

经过光栅化、顶点着色器与片元着色器的介绍，可以看出顶点着色器每顶点一执行，而片元着色器每片元一执行，片元着色器的执行次数明显大于顶点着色器的执行次数。因此在开发中，应尽量减少片元着色器的运算量，将一些复杂运算尽量放在顶点着色器中执行。

3.2.3　OpenGL ES中立体物体的构建

前面的几个小节向读者介绍了不同版本OpenGL ES的渲染管线，同时也给出了一个非常简单的三角形案例。但到目前为止，读者可能还是不太清楚虚拟3D世界中的立体物体是如何搭建出来的。其实这与现实世界搭建建筑物并没有本质区别，请读者考察图3-25和图3-26中中国国家大剧院远景和近景的照片。

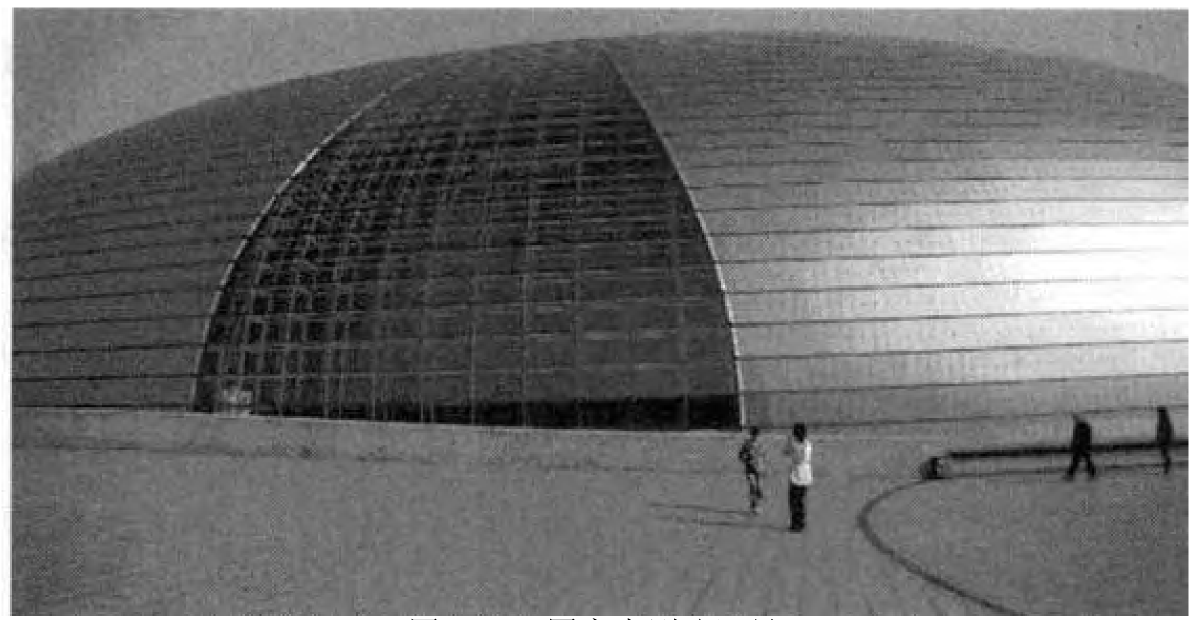


图3-25　国家大剧院远景

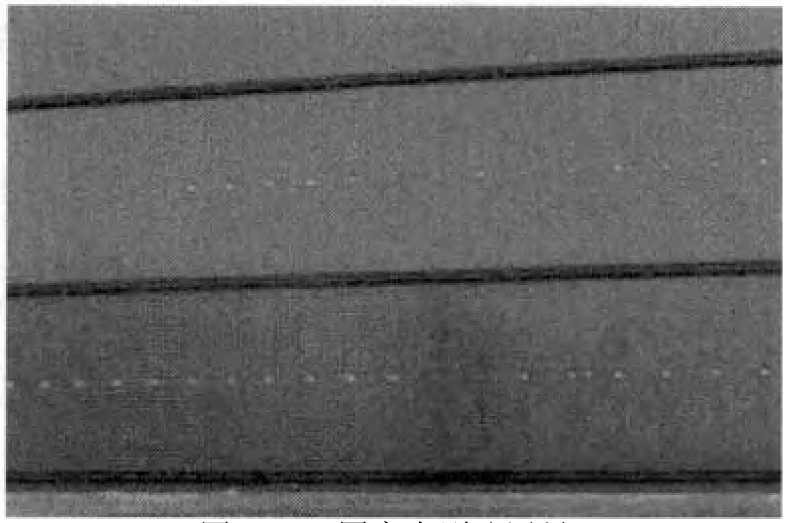


图3-26　国家大剧院近景

从两幅照片中可以对比出，现实世界的某些建筑物远看是平滑的曲面，其实近看是由一个一个的小平面组成的。3D虚拟世界中也是如此，任何立体物体都是由多个小平面搭建而成的。这些小平面切分得越小，越细致，搭建出来的物体就越平滑。

当然OpenGL ES的虚拟世界与现实世界还是有区别的，现实世界中可以用任意形状的多边形来搭建建筑物，例如图3-25中的国家大剧院就是用四边形搭建的，而OpenGL ES中仅仅允许采用三角形来搭建物体。其实这从构造能力上来说并没有区别，因为任何多边形都可以拆分为多个三角形，只需开发时稍微注意一下即可。

提示

OpenGL ES中之所以仅仅支持三角形而不支持任意多边形是出于性能的考虑，就目前移动嵌入式设备的硬件性能情况来看，这是必然的选择了。

图3-27所示更加具体地说明了，在OpenGL ES中如何采用三角形来构建立体物体。

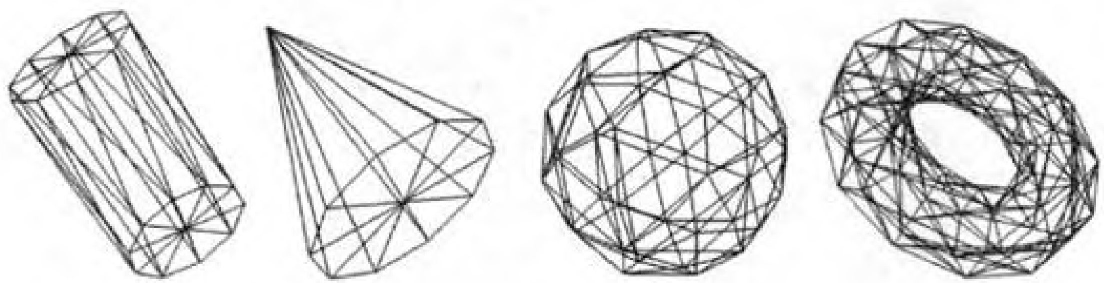


图3-27　用三角形搭建立体物体



说明

从图3-27中可以看出用三角形可以搭建出任意形状的立体物体，这里仅仅是给出了几个简单的例子，本书的后继章节中还有很多其他形状的立体物体。

了解了OpenGL ES中立体物体的搭建方式后，下面就需要了解OpenGL ES中的坐标系统了。OpenGL ES中采用的是三维笛卡尔坐标系，如图3-28所示。

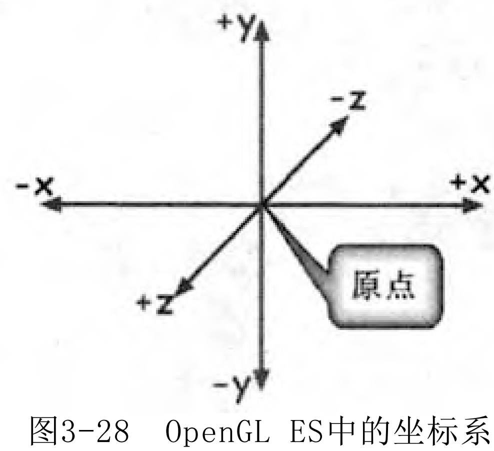


图3-28　OpenGL ES中的坐标系

从图3-28中可以看出，OpenGL ES中采用的是左手标架坐标系。一般来说，初始情况下*Y*轴平行于屏幕的竖边，*X*轴平行于屏幕的横边，*Z*轴垂直于屏幕平面。

> **提示**
>
> 空间解析几何中有两种坐标系标架，左手标架和右手标架。本书非讨论空间解析几何的专门书籍，关于标架的问题不予详述，需要的读者可以参考空间解析几何的相关书籍或资料。

3.3　主流Android手机GPU大PK

GPU一词是由英伟达（NVIDIA）在1999年提出来的，其相当于专用于图像处理的CPU，使用GPU在处理图像时的工作效率会远高于通用CPU。在PC时代，带有优良性能GPU的显卡是游 戏玩家在购买PC时主要考察的方面之一。

在智能手机硬件飞速发展的今天，1GHz双核处理器已经成为高端智能手机的标准配置，目前已经有多款搭载1GHz以上的双核处理器的Android智能手机上市。在处理器效率的提升以及3D手机游戏的带动下，手机内置GPU的性能也成为了一个衡量手机性能的重要标准。

3.3.1　手机GPU四大家族对比

Android平台中，由于没有统一的硬件标准，导致了各厂家，各型号手机的硬件配置大不相同。目前应用在手机和平板电脑端的GPU主要由4家公司执行设计——英伟达、Imagination公司、高通公司和ARM公司。其中，ARM和Imagination公司并不进行GPU的实际生产，主要出售芯片设计技术（IP）的授权。

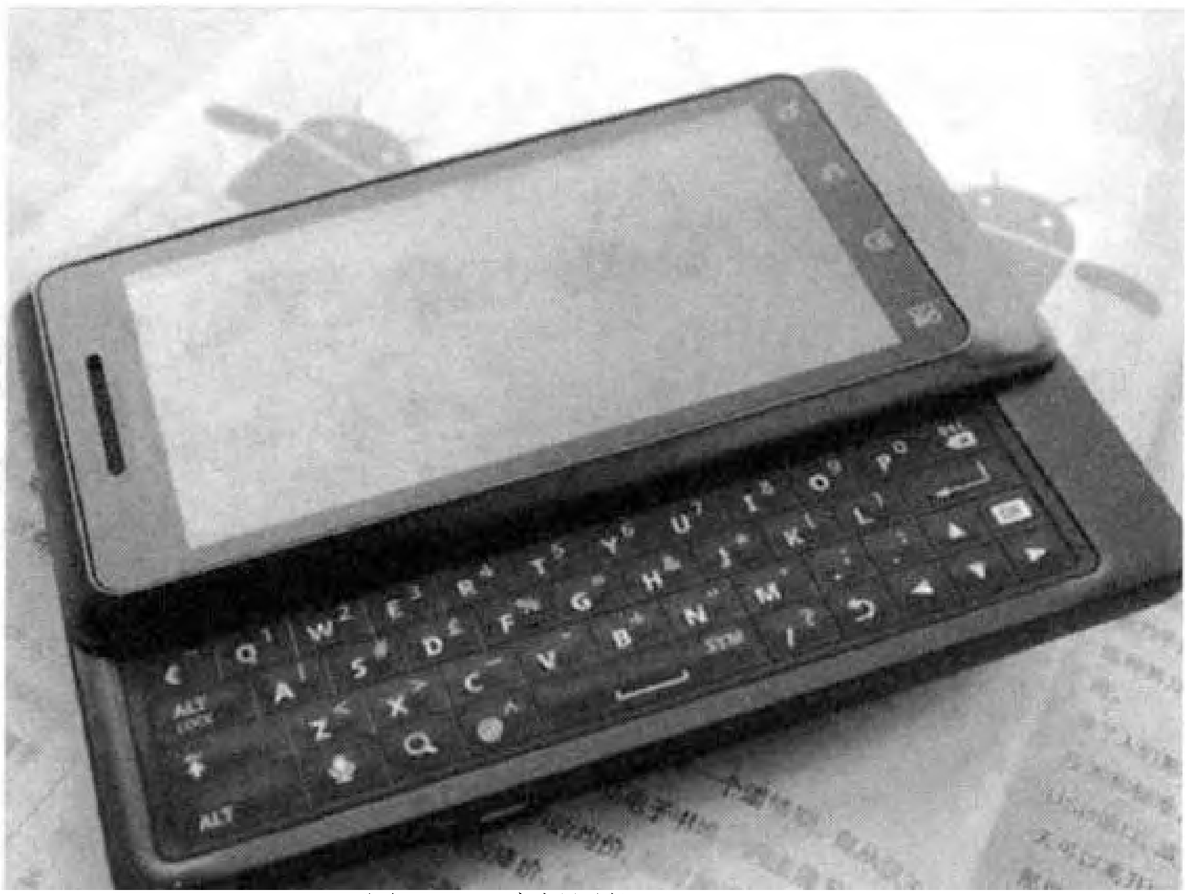
不同于PC的独立显卡，智能手机的GPU是被整合到CPU里面的（两者在同一个芯片里面），因此在提及GPU的时候，其主体的CPU也必须要涉及，下面将对这4家公司出品的GPU逐一进行介绍。

1．PowerVR SGX系列

Imagination公司以著名的PowerVR SGX系列为主，主要型号为PowerVR SGX530、PowerVR SGX535、PowerVR SGX540、PowerVR SGX545及PowerVR SGX543。SGX系列芯片支持的3D API包括OpenGL

ES、DirectX、SM3.0，目前上市的型号都支持可编程渲染管线。著名的三星的S5PC100/C110/S5PV210、苹果的A4/A5等芯片都是采用此系列。

- PowerVR SGX530是被广泛地采用在TI和三星的SoC芯片上的3D图形加速器。采用PowerVR SGX530的经典手机摩托罗拉Milestone 2（里程碑2）就是其中的代表产品，如图3-29所示。


图3-29　摩托罗拉Milestone2

- PowerVR SGX535一开始被应用在苹果的3GS上，随着苹果iPad和iPhone 4的发布，该GPU和A4芯片一起被使用在苹果的多款掌上设备上。PowerVR SGX535的三角形生成率2倍于PowerVR SGX530，像素填充率也有1.25倍的提升。
- PowerVR SGX540实际的性能比PowerVR SGX535提升2倍以上，被大量应用在三星的很多产品里。最流行的如三星的19000，其外观如图3-30所示。



图3-30　三星19000智能手机

- PowerVR SGX545支持DirectX 10.1和OpenGL 3.0技术，支持MEPG-4、VC-1、WMV9、H.264等格式的高清视频解码。根据其图形核心频率分为高频版和低频版两款，分别为600/400MHz，可支持HDMI 1.3a和DisplayPort 1.1输出标准。
- PowerVR SGX543采用多指令，多数据流的架构，可采用多种核心配置，从双核、四核到八核直至十六核（分别标记为SGX543 MP2、SGX543 MP4、SGX543 MP8、SGX543 MP16）。与前面介绍的SGX系列不同，SGX543使用的是统一渲染架构，此架构的采用成为其性能明显强于当前市面上任何嵌入式GPU的有力保证。

2．Mali系列

ARM公司设计出品的GPU是Mali系列，主要型号为Mali-55、Mali-200、Mali-300、Mali-400、Mali-T604等，其授权的厂商较多，但是在高端智能手机、平板电脑上的应用很少见。一般用在低端平板上，如最近较火的智器T10、蓝魔W10就是搭载的整合了Mali-400型号GPU的AMLogic AML8726-M型号的CPU。

- Mali-55核心。

目前ARM Mali系列当中，Mali-55是最低档的一款GPU，其符合OpenGL ES 1.x和OpenVG 1.0（矢量图形算法加速）标准。在275MHz的频率下，拥有1Mtri/s（三角形生产率）和100Mpix/s（像素填充率）的处理能力。采用Mali-55方案的MID产品有蓝魔W9和台电T720，分别如图3-31和图3-32所示。



图3-31　蓝魔W9



图3-32　台电T720

说明

从芯片的定位和实际情况来看，其主要设计理念是以最小的芯片面积来提供高质量的图形处理性能，并保证系统的负荷低、能耗低。因此随着其的出现，主要针对的是GPS导航、MP4等便携设备，这些设备的分辨率不高，应用也不需要过于专业。

- Mali-200核心。

相比Mali-55力求基本满足便携设备的需要，Mali-200定位要高很多，其主要为PMP、PND、家庭和车载音响以及包括手机在内的便携式消费产品服务。为这些设备提供先进的2D和3D图形处理能力，并保证较小的内存带宽占用，如图3-33所示的酷比魔方U6便是使用的该核心。



图3-33　酷比魔方U6

说明

如果说作为入门级的图形核心，Mali-55运行一些3D特效和普通3D游戏问题不大的话，那么对于需要支持OpenGL ES 2.0的大型3D游戏来说，就必须要达到Mali-200核心以上了。

- Mali-400 MP核心。

Mali-400 MP是世界上第一个符合OpenGL ES 2.0的多核GPU，可提供卓越的二维和三维图形加速性能，最高可扩展至1080p分辨率，同时使ARM® 在能耗和带宽效率方面保持领先地位。如图3-34所示的蓝魔W10采用的便是Mali-400 MP核心。

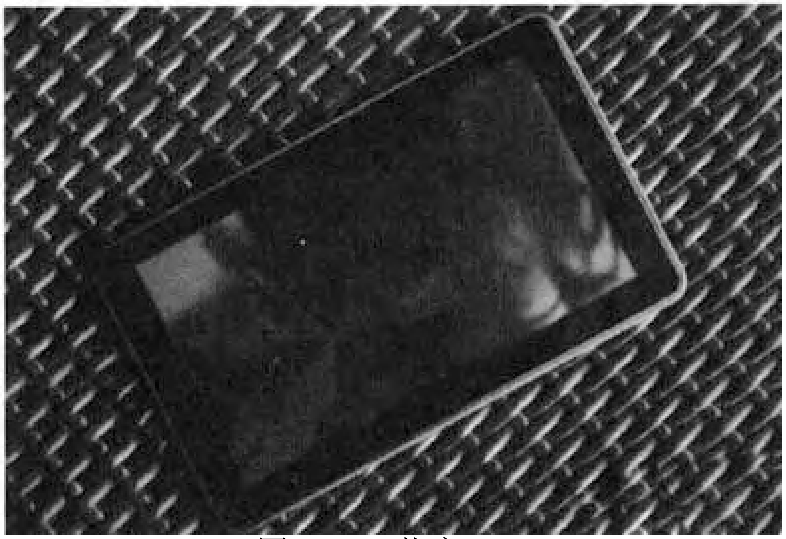


图3-34　蓝魔W10

- Mali-300核心。

因为Mali-300的推出时间比Mali-400要晚，所以把Mali-300放在后面来介绍。Mali-300 GPU提供的性能水平介于ARM Mali-200和可扩展地多核Mali-400嵌入式图形解决方案之间。Mali-300 GPU的基础架构与Mali-400相同，拥有8KB二级高速缓存，能以业界领先的存储器带宽效率实现高清性能。

- Mali-T604核心。

Mali-T604属于ARM推出的最新的第四代嵌入式图形芯片。该芯片使用了创新性的三管道设计，核心数量可以配置至最多4核，其最高性能可达到第三代产品的5倍。

3．Adreno系列

高通的产品主要是Adreno系列，其被广泛应用于高通的Snapdragon平台上。Adreno GPU至今已经发布了几款产品，在移动设备中应用也比较广泛。这些产品包括：Adreno 130、Adreno 200、Adreno 205、Adreno 220。目前的主流是Adreno 130、Adreno 200以

及高端的Adreno 205。下面将对Adreno系列各个型号的GPU进行简要的介绍。

- Adreno 130核心。

Adreno 130是第一款Adreno图形处理器，可以提供完整的3D图形硬件支持。Adreno 130 GPU支持的图形标准为OpenGL ES 1.1固定渲染管线；三角形输出率是4M/s，像素填充率是133M/s。图3-35所示的为采用Adreno 130 GPU的手机产品。



图3-35　采用Adreno 130 GPU的手机

- Adreno 200核心。

Adreno 200在OpenGL ES标准下的图形处理质量上有了显著提升，开发人员可以最大限度地使用其图形处理能力。其支持的图形标准为OpenGL ES 2.0、OpenGL ES 1.x等。此核心GPU图形性能为三角形输出率22M/s，像素填充率133M/s。HTC的著名机型G7便是使用的Adreno 200 GPU，如图3-36所示。



图3-36　经典HTC G7智能手机

- Adreno 205核心。

Adreno 205提供了更强的2D和3D处理能力，达到了Adreno 200的四倍，却拥有更低的功耗，支持的图形标准为OpenGL ES 2.0、OpenGL ES 1.1、OpenVG 1.1等。三角形输出率为41M/s，像素填充率为245M/s。HTC Desire Z便是采用的该GPU，如图3-37所示。



图3-37 HTC Desire Z手机

- Adreno 220核心。

Adreno 220于2011年推出，相比Adreno 205有几倍性能的提升，在游戏、导航及浏览器应用 中的表现出色，而且还可提供更大的显示分辨率支持和更低的电源能耗。图形性能方面，三角形输出率为88M/s，像素填充率为532M/s。HP TouchPad便采用了该GPU，如图3-38所示。


图3-38　HP TouchPad

4．GeForce ULV系列

英伟达在智能移动设备上的GPU主要是搭载Tegra 2的GeForce ULV，代表机型为摩托罗拉的ATRTX 4G。在2011 MWC上，英伟达展示了Tegra 3在Android设备上运行的情况，并透露了他们未来的一些发展蓝图。从性能上来看，英伟达的GeForce系列图形芯片在整体上非常优秀，特别是在1080P视频录制和播放方面有着出色的表现。

- Tegra2核心。

CES2010大展上，英伟达正式发布1GHz频率的双核处理器Tegra 2芯片，摩托罗拉Atrix 4G作为首批搭载该芯片的智能手机，如图3-39所示。

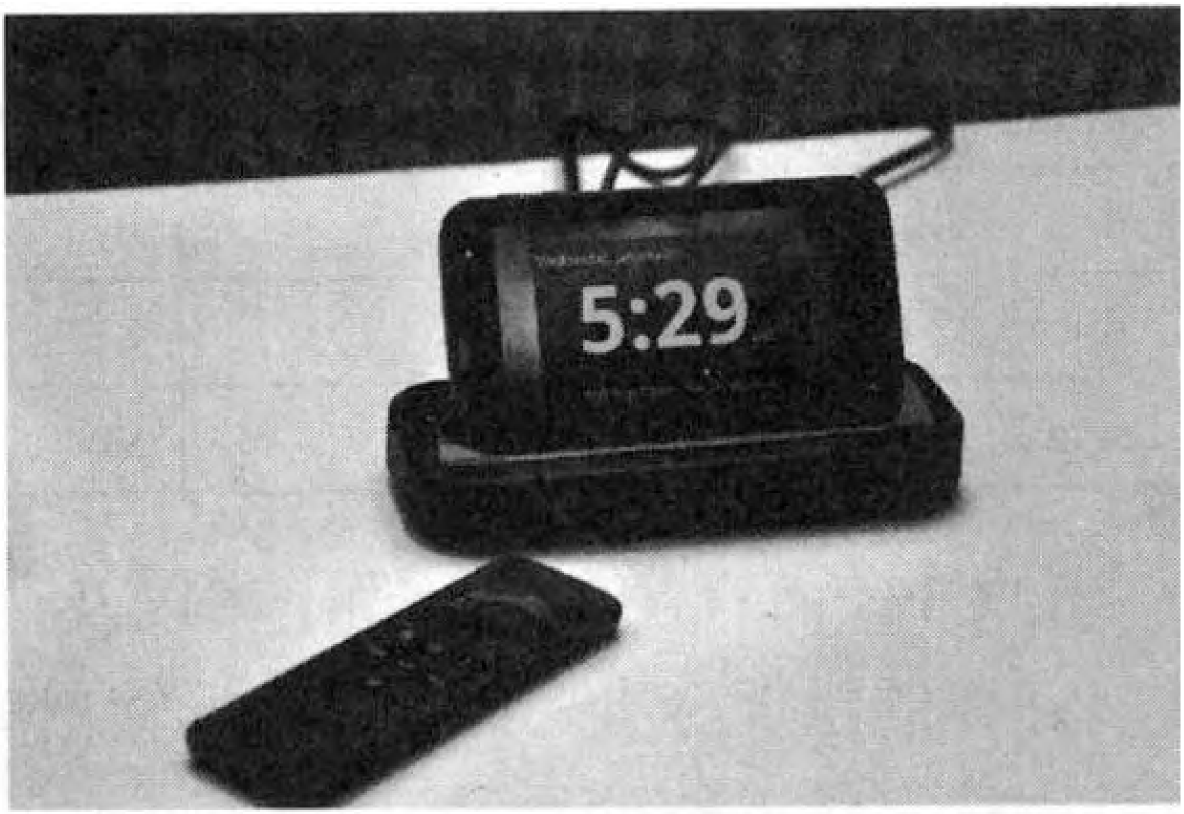


图3-39　摩托罗拉Atrix 4G

说明

图形处理方面，在已知参数上得知，Tegra2在图形处理方面配置了独立的“显卡”，为最新的Geforce 6芯片，支持Open GL ES2.0/1.1, Open VG标准，比第一代Tegra芯片的图形处理速度提高2~3倍。在实际操作中体现为，Atrix 4G在运行大型3D游戏方面的表现非常抢眼。

- Tegra 3核心。

Tegra 3是集成12个执行单元的GeForce GPU图形核心，支持3D立体，并支持2560×1600的超高清分辨率。其整体性能号称5倍于Tegra2（CPU处理能力比上代Tegra 2提升了两倍以上，而GPU图形性能比上代提升了3倍），Tegra 3可能会首先搭载在摩托罗拉的手机上上市。

3.3.2　主流GPU的性能参数比较

前一小节对手机GPU的4大家族做了较为详细的介绍，相信读者对PowerVR SGX系列、Mali系列、Adreno系列以及GeForce ULV系列有了一定的了解。接下来在本小节中将对比各个系列的主要性能参数（三角形输出率以及像素填充率），具体情况如表3-1所列。

表3-1　各个系列GPU参数对比

GPU型号	三角形输出率（M/s）	像素填充率（M/s）
PowerVR SGX530	14	125
PowerVR SGX535	28	400
PowerVR SGX540	35	1000
PowerVR SGX545	60	1000
PowerVR SGX543	133	4000
Mali-55	1	100
Mali-200	16	275
Mali-300	30	275
Mali-400 MP	30	275
Adreno 130	4	133
Adreno 200	22	133
Adreno 205	41	245
Adreno 220	88	532
Tegra2	90	1200

对GPU而言，三角形输出率以及像素填充率这两个性能参数显得尤为重要，是衡量一款GPU性能好坏的重要标准。其中三角形输出率指的是单位时间内输出三角形的数量，像素填充率指的是每秒钟填充像素的数量。

提示

对于Mali-T604和Tegra3，ARM公司与英伟达公司尚未给出明确的性能参数说明，所以在各个系列GPU参数对比时并未进行介绍。另外本节所列举的GPU型号均为截止到作者完稿时已经上市的型号，在硬件飞速发展的今天，读者买到这本书时，可能已经出现了更新型号、更高性能的GPU。

3.4　本章小结

随着Android手机硬件的发展以及OpenGL ES图形编程标准的逐渐完善，编程人员可以使用OpenGL ES 2.0开发出更加绚丽的3D场景，从而满足日益增长的软件需求。通过本章的学习，读者应该对OpenGL ES 2.0的渲染管线有一定的了解，学会简单3D场景的开发，并对当前手机的GPU型号与其所支持的API有一定的了解，为以后开发酷炫的3D游戏打下坚实的基础。

第4章　着色语言 Shading Language

由于当下可编程图形硬件占图形硬件市场的份额越来越大，3D开发人员应该具备使用可编程图形硬件的能力，而各种可编程图形硬件一般都是提供着色器供开发人员使用的，因此学习着色语言是开发3D游戏的重中之重。

由于Android平台下的可编程图形硬件支持的是OpenGL ES 2.0标准，因此，本章将向读者介绍OpenGL ES着色语言。OpenGL ES着色语言是基于OpenGL ES 2.0平台研发的，基本可以看做是PC和工作站上OpenGL着色语言的子集，包含了OpenGL着色语言的大部分功能。

4.1 着色语言概述

OpenGL ES着色语言是一种高级的图形编程语言，其源自应用广泛的C语言，同时具有RenderMan以及其他着色语言的一些优良特性，易于被开发人员掌握。

与传统通用编程语言有很大不同的是，其提供了更加丰富的原生类型，如向量、矩阵等。这些新特性的加入使得OpenGL ES着色语言在处理3D图形方面更加高效、易用。简单来说，OpenGL ES着色语言主要包括以下特性。

- OpenGL ES着色语言是一种高级的过程语言（注意，不是面向对象）。
- 对顶点着色器、片元着色器使用的是同样的语言，不做区分。
- 基于C/C++的语法及流程控制。
- 完美支持向量与矩阵的各种操作。
- 通过类型限定符来管理输入与输出。
- 拥有大量的内置函数来提供丰富的功能。

总之，OpenGL ES着色语言是一种易于实现、功能强大、便于使用、完美支持硬件灵活性，并且可以高度并行处理、性能优良的高级图形编程语言。其可以帮助开发人员在不浪费大量时间的情况下，轻松地为用户带来更完美的视觉体验，开发出更加酷炫的3D场景与特效。

对于3D游戏开发人员来说，掌握这门语言尤为重要。本章的后继内容将从多个方面介绍OpenGL ES着色语言的基本知识，使得读者初步了解着色语言，为以后深入地学习打下坚实的基础。

4.2 着色语言基础

OpenGL ES着色语言虽然是基于C/C++语法的语言，但是其与C/C++相比较还是有很大不同的。例如，该语言不支持双精度浮点型（double）、字节型（byte）、短整型（short）、长整型（long），并且取消了C中的联合体（union）、枚举类型（enum）、无符号整型（unsigned）以及位运算等特性。

4.2.1 数据类型概述

与C语言类似，着色语言中有许多内建的原生数据类型以及构建数据类型，如：浮点型（float）、布尔型（bool）、整型（int）、矩阵型（matrix）以及向量型（vec2、vec3等）等。总体来说，这些数据类型可以分为标量、向量、矩阵、采样器、结构体以及数组等几类。本小节将为读者简单介绍这些数据类型，包括其基本知识与基本使用方法。

1．标量

标量也被称为“无向量”，其值只具有大小，并不具有方向。标量之间的运算遵循简单的代数法则，如质量、密度、体积、时间以及温度等都属于标量。OpenGL ES着色语言支持的标量类型有bool、int与float，各自的用法如下所列。

- 布尔型——bool。

布尔型用来声明一个单独的布尔数，其值只能为true与false中的一个。布尔类型的值一般由关系运算或者逻辑运算产生，在着色语言的流程控制中只能使用布尔类型的值作为表达式，其基本用法如下所示。

```
bool b;                                   //声明一个布尔型的变量
```

- 整型——int。

整型用来声明一个单独的整数，其值可以为正数、负数及0。与C语言中的整型有所不同，在OpenGL ES着色语言中的整数保证最少支持16位精度。开发时注意运算需要在正确的范围内进行（若16位精度范围为[-65535～65536]，着色语言中所谓的精度不含符号位），超出运算范围可能产生溢出问题。

OpenGL ES着色语言中整数也可以像C语言中一样，用十进制、八进制或者十六进制等不同的进制表示，基本用法如下所示。

```
int a = 15;                 //十进制
int b = 036;                //0开头的字面常量为八进制,代表十进制的30
int c = 0x3D;               //0x开头的字面常量为十六进制，代表十进制的58
```

- 浮点型——float。

浮点型用来声明一个单独的浮点数，基本用法如下所示。

```
float f;                    //声明一个float型的变量
```

```
float g = 2.0;              //在声明变量的同时为变量赋初值
float h, i;                 //同时声明多个变量
float j, k = 2.56, l;       //声明多个变量时，可以为其中某些变量赋初值
```

需要注意的是，由于OpenGL ES着色语言中没有像C/C++语言一样提供多种不同精度的整数与浮点数，因此代码中的字面常量就不需要使用后缀来说明精度了，只要给出值即可。

2．向量

OpenGL ES着色语言中，向量可以看做是用同样类型的标量组成的，其基本类型也分为bool、int及float 3种。每个向量可以由2个、3个或者4个相同的标量组成，具体情况如表4-1所列。

表4-1　各向量类型及说明

向量类型	说　明	向量类型	说　明
vec2	包含了2个浮点数的向量	ivec4	包含了4个整数的向量
vec3	包含了3个浮点数的向量	bvec2	包含了2个布尔数的向量
vec4	包含了4个浮点数的向量	bvec3	包含了3个布尔数的向量
ivec2	包含了2个整数的向量	bvec4	包含了4个布尔数的向量
ivec3	包含了3个整数的向量		

声明向量类型变量的基本语法如下。

```
vec2 v2;                    //声明一个vec2类型的向量
ivec3 v3;                   //声明一个ivec3类型的向量
bvec4 v4;                   //声明一个bvec4类型的向量
```

向量在着色器代码的开发中有着十分重要的作用，可以很方便地存储以及操作颜色、位置、纹理坐标等不仅包含一个组成部分的量。开发中，有时也可能需要单独访问向量中的某个分量，基本的语法为“〈向量名〉•〈分量名〉”，根据目的的不同，主要有如下几种用法。

- 将一个向量看作颜色时，可以使用r、g、b、a4个分量名，其分别代表红、绿、蓝、透明度4个色彩通道，具体用法如下。

```
aColor.r=0.6;                    //给向量 aColor 的红色通道分量赋值
aColor.g=0.8;                    //给向量 aColor 的绿色通道分量赋值
```

> **提示**
>
> 若向量是4维的，则可以使用的分量名为：r、g、b、a；若向量为3维的，则可以使用的分量名为r、g、b；若为2维的，则可以使用r、g。

- 将一个向量看作位置时，可以使用x、y、z、w等4个分量名，其分别代表*X* 轴、*Y* 轴、*Z* 轴、向量的模等4个分量，具体用法如下。

```
aPosition.x=67.2;                //给向量 aPosition 的 X 轴分量赋值
aPosition.z=48.3;                //给向量 aPosition 的 Z 轴分量赋值
```

> **提示**
>
> 若向量是4维的，则可以使用的分量名为：x、y、z、w;若向量为3维的，则可以使用的分量名为x、y、z；若为2维的，则可以使用x、y。

- 将一个向量看作纹理坐标时，可以使用s、t、p、q等4个分量名，其分别代表纹理坐标的不同分量，具体用法如下。

```
aTexCoor.s=0.65;                    //给向量 aTexCoor 的 s 分量赋值
aTexCoor.t=0.34;                    //给向量 aTexCoor 的 t 分量赋值
```

提示

若向量是4维的，则可以使用的分量名为：s、t、p、q；若向量为3维的，则可以使用的分量名为s、t、p；若为2维的，则可以使用s、t。

访问向量中的各个分量不但可以采用“.”加上不同的分量名，还可以将向量看作一个数组，用下标来进行访问，具体用法如下。

```
aColor[0]=0.6;                      //给向量 aColor 的红色通道分量赋值
aPosition[2]=48.3;                  //给向量 aPosition 的 z 轴分量赋值
aTexCoor[1]=0.34;                   //给向量 aTexCoor 的 t 分量赋值
```

提示

关于纹理坐标中的s、t分量等读者可能不是很明白，这不用担心，本书后面介绍纹理贴图的章节会进行详细的介绍。

其实，在C语言中也可以通过自己构建结构体的方式来支持向量，但进行向量的运算时必须由CPU将每个分量依次顺序计算（3个分量就需要计算3次），效率不高。而着色器中的向量则不同，其由硬件原生支持，进行向量的运算时也是各分量并行一次完成的（n个分量只需要一次计算），效率大大提高。

3．矩阵

有一些基础的开发人员都知道，3D场景中的移位、旋转、缩放等变换都是由矩阵的运算来实现的。因此3D场景的开发中会非常多地使

用到矩阵，因此OpenGL ES着色语言中也提供了对矩阵类型的支持。这大大方便了开发，免去了自行构建矩阵的麻烦。

矩阵按尺寸分为2×2矩阵、3×3矩阵以及4×4矩阵，具体情况如表4-2所列。

表4-2　矩阵的类型及说明

矩阵类型	说　明
mat2	2×2的浮点数矩阵
mat3	3×3的浮点数矩阵
mat4	4×4的浮点数矩阵

矩阵类型的基本用法如下。

```
mat2 m2;                  //声明一个mat2类型的矩阵
mat3 m3;                  //声明一个mat3类型的矩阵
mat4 m4;                  //声明一个mat4类型的矩阵
```

OpenGL ES着色语言中，矩阵是按列顺序组织的，也就是一个矩阵可以看作由几个列向量组成。例如，mat3就可以看作由3个vec3组成。

对于矩阵的访问，可以将矩阵作为列向量的数组来访问。如：matrix为一个mat4，可以使用matrix[2]取到该矩阵的第3列，其为一个vec4；也可以使用Matrix[2][2]取得第3列的向量的第3个分量，其为一个float；其他的依此类推。

提示

从数学上讲，矩阵看作由向量组成时有两种选择：可以将矩阵看作由多个行向量组成或看作由多个列向量组成。虽然不同的选择功能一样，但在具体进行计算时是有所不同的，因此了解OpenGL ES

的选择非常重要。对这方面数学不太了解的读者可以参看一些线性代数的资料。

4．采样器

采样器是着色语言中不同于C语言的一种特殊的基本数据类型，其专门用来进行纹理采样的相关操作。一般情况下，一个采样器变量代表一幅或一套纹理贴图，其具体情况如表4-3所列。

表4-3　采样器基本类型及说明

采样器类型	说　明	采样器类型	说　明
sampler2D	用于访问二维纹理	samplerCube	用于访问立方贴图纹理
sampler3D	用于访问三维纹理		

需要注意的是，与前面介绍的几类变量不同，采样器变量不能在着色器中初始化。一般情况下采样器变量都用uniform限定符来修饰，从宿主语言（如Java）接收传递进着色器的值。

提示

sampler3D并不是在所有的OpenGL ES实现中都支持，因此，若要使用必须首先在着色器代码中进行设置，打开相应的扩展。

5．结构体

OpenGL ES着色语言还提供了类似于C语言中的用户自定义结构体，同样也是使用struct关键字进行声明，基本用法如下所示。

```
struct info{                        //声明一个结构体info
    vec3 color;                     //颜色成员
    vec3 position;                  //位置成员
    vec2 textureCoor;               //纹理坐标成员
}
```

声明了info类型的结构体之后，就可以像使用内建数据类型一样使用这个用户自定义的类型了，如：

```
info CubeInfo;                      //声明了一个info类型的变量CubeInfo
```

> **提示**
>
> 对于结构体来说，其他的使用方法都与C语言基本类似，笔者就不再赘述，读者可自行查阅并参考C语言的结构体部分的相关知识。

6．数组

从前面向量的介绍中读者就会感觉到，着色语言应该是支持数组的。在着色语言中，开发人员可以声明任何类型的数组。声明数组的方式主要有两种，具体如下所列。

（1）在声明数组的同时，指定数组的大小：

```
vec3 position[20];                  //声明了一个包含20个vec3的数组，索引从0开始。
```

（2）在声明数组时，也可以不指定数组的大小，但是必须符合下列两种情况之一。

- 引用数组之前，要再次使用第一种声明方式来声明该数组，代码如下。

```
vec3 position[];                  //声明了一个大小不定的vec3型数组。
vec3 position[5];                 //再次声明该数组，并且指定大小。
```

提示

再次声明数组，指定大小之后，就不能再进行声明了。

- 代码中访问数组的下标都是编译时常量（如：字面常量），这时编译器会自动创建适当大小的数组，使得数组尺寸足够存储编译器看到的最大索引值对应的元素，代码如下。

```
1 vec3 position[];                  //声明了一个大小不定的vec3型数组
2 position[3] = vec3(3.0);          //position需要为一个大小为4的数组
3 position[20] = vec3(6.0);         //position需要为一个大小为21的数组
```

说明

上述代码的第2行需要尺寸为4的数组，代码的第3行需要尺寸为21的数组，若后面没有更大的下标出现，则编译器自动创建position为尺寸21的数组。这种由系统量体裁衣、自动分配的方法若恰当采用可以提高硬件的使用效率，降低对硬件资源的消耗。

7. 空类型

空类型使用void表示，仅用来声明不返回任何值的函数。例如在顶点着色器以及片元着色器中必须存在的main函数就是一个返回值为空类型的函数，代码如下所示。

```
void main()                       //声明一个空返回值类型的main方法
{  //函数的具体操作省略 ……  }
```

4.2.2 数据类型的基本使用

前一小节介绍了OpenGL ES着色语言中的各个数据类型，掌握了这些数据类型的基本知识后，本小节将为读者简单地介绍这些数据类型的声明、初始化以及作用域的问题。

1．声明、作用域及初始化

变量的声明及作用域与C++语法类似，可以在任何需要的位置声明变量，同时其作用域也与C++类似，分为局部变量与全局变量。请读者考察如下代码片段。

```
1 int a,b;                                //声明了全局变量a及b
2 vec3 aPosition=vec3(1.0,2.0,3.3);       //声明了全局变量aPosition并赋初值
3 void myFunction()
4 {
5     int c=14;                           //声明了局部变量c并赋初值
6     a=4;                                //给全局变量a赋值
7     b=a*c;                              //给全局变量b赋值
8 }
```

- 第1行声明了全局变量a、b,其作用域为整个着色器。
- 第5行声明了局部变量c，其作用域为自其声明开始到myFunction函数结束。

> **提示**
>
> 着色语言还有一点特殊的地方，在一些着色语言的实现中不可以在if语句中声明新的变量，这是为了简化变量在else子句上作用域的实现。笔者目前使用的Android平台的各种实现基本都是支持在if语句中声明变量的，但也不排除有特殊情况出现，读者了解这一点即可。

虽然着色器中变量的命名很自由，仅要求变量由字母、数字与下划线组成，且必须以字母或者下划线开头。但开发人员在开发程序时，有一个良好的命名习惯将大大提高代码的可维护性，同时使得开发的代码简洁、美观，可读性强。因此，笔者建议在OpenGL ES着色语言中，变量按照如下规则进行命名。

- 由于系统中有很多内建变量都是以“gl_”作为开头，因此用户自定义的变量不允许使用“gl_”作为开头。
- 为自己的函数或变量取名时尽量采用有意义的拼写，除了一些局部变量外不要采用a、b、c这样的名称。若一个单词不足以描述变量的用途，可以用多个单词组合，除第一个单词全小写外，其他每个单词的第一个字母大写。

向量的初始化还有一些灵活变化的技巧，下面的代码片段说明了这个问题。

```
1 float a=12.3;               //声明了浮点变量 a 并赋初值
2 float b=11.4;               //声明了浮点变量 b 并赋初值
3 vec2 va=vec2(2.3,2.5);      //声明了 2 维向量 va 并赋初值
4 vec2 vb=vec2(a,b);          //声明了 2 维向量 vb 并赋初值
5 vec3 vc=vec3(vb,13.5);      //声明了 3 维向量 vc 并赋初值
6 vec4 vd=vec4(va,vb);        //声明了 4 维向量 vd 并赋初值
7 vec4 ve=vec4(0.2);          //声明了 4 维向量 ve 并赋初值，相当于 vec4(0.2,0.2,0.2,0.2)
```

说明

从上述代码中可以看出，初始化时向量的各个分量既可以使用字面常量，也可以使用变量，还可以从其他向量直接获取。同时，若向量各分量的值相同，还可以采用如第7行代码所示的简化语法。实际开发中，读者可以根据具体情况灵活选用。

矩阵的初始化也有一些灵活变化的技巧，下面的代码片段说明了这个问题。

```
1 float a=6.3;                                   //声明了浮点变量a并赋初值
2 float b=11.4;                                  //声明了浮点变量b并赋初值
3 float c=12.5;                                  //声明了浮点变量c并赋初值
4 vec3 va=vec3(2.3,2.5,3.8);
5 vec3 vb=vec3(a,b,c);
6 vec3 vc=vec3(vb.x,va.y,14.4);
7 mat3 ma=mat3(1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,c);//通过给出9个字面常量初始化3x3的矩阵
8 mat3 mb=mat3(va,vb,vc);                        //通过给出3个向量初始化3x3的矩阵
9 mat3 mc=mat3(va,vb,1.0,2.0,3.0);               //通过给出2个向量和3个字面常量初始化3x3的矩阵
10    mat3 md=mat3(2.0) ;                        //通过给出1个字面常量初始化3x3的矩阵
```

说明

从上述代码中可以看出，初始化时矩阵的各个元素既可以使用字面常量，也可以使用变量，还可以从其他向量直接获取。同时，若矩阵只有对角线上有值且相同，还可以采用如第10行代码所示的简化语法进行初始化，其等同于mat3（2.0，0.0，0.0，0.0，2.0，0.0，0.0，0.0，2.0）。实际开发中，读者可以根据具体情况灵活选用。

2．变量初始化的规则

该语言中的变量初始化规则基本承袭自C语言，但也有一些不同，基本规则如下所列。

- 常用初始化方式：变量可以在声明的时候就进行初始化。

```
int a=2,b=3,c;                    //声明了int型的变量a、b与c，同时为a与b变量赋初值
```

- 用const限定符修饰的变量必须在声明的时候进行初始化。

```
2 const float k=1.0;                    //在声明的时候初始化
```

- 属性变量、一致变量以及易变变量在声明的时候一定不能进行初始化。

```
1 attribute float angleSpan;            //不可对属性变量进行初始化
2     uniform int k;                    //不可对一致变量进行初始化
3 varying vec3 position;                //不可对易变变量进行初始化
```

> **提示**
>
> 关于属性变量、一致变量以及易变变量的细节在后面会进行详细的介绍，这里读者简单了解语法规则即可。

4.2.3　运算符

前两小节为读者介绍了OpenGL ES着色语言的各种数据类型及其用法，本小节将介绍如何实现对各种数据的操作，这在开发中是十分重要的。

与大多数编程语言类似，常见的运算符都可以在该语言中使用。下面按照优先级顺序列出了OpenGL ES着色语言中可以使用的运算符，如表4-4所示。

表4-4　运算符列表（按照优先级顺序排列）

运算符	说明	运算符	说明
[]	用于索引	.	用于成员选择与混合
++ --	自加1与自减1 后缀	++ --	自加1与自减1 前缀
- !	一元非及逻辑非	* /	乘法与除法
+ -	加法与减法	<> <= >=	关系运算符
== !=	等于以及不等于	&&	逻辑与
^^	逻辑异或	||	逻辑或
?:	选择	= += -= *= /=	赋值运算符

提示

表4-4中的优先级按照先从左到右，再从上到下的顺序。

了解了运算符的优先级之后，下面对常用的一些运算符进行一下更为详细的介绍。

1．索引

从表4-4中可以发现，着色语言中的索引表示方法与C语言中是完全相同的，用“[]”来表示。并且在OpenGL ES着色语言中，索引的起始下标也为0。

索引经常用在对数组、向量或者矩阵的操作中，通过索引操作，可以方便地获取数组、向量或者矩阵中包含的各种元素，具体使用如下面的代码片段所示。

```
float array[10];              //声明一个大小为10的float型数组
array[2]=1.0;                 //通过索引操作拿到下标为2的数，并且给其赋值
vec3 position=vec3(2.3,5.0,0.2);//声明一个vec3类型的向量，并且进行初始化
float temp=position[1];       //通过索引对position向量进行操作，拿到其第二个值5.0并赋给temp
mat4 matrix=mat4(1.0);        //声明一个mat4类型的矩阵，并进行初始化
vec4 tempV=matrix[1]; //通过索引对matrix矩阵进行操作，拿到向量(0.0,1.0,0.0,0.0)并赋值给tempV
```

2．混合选择

通过运算符“·”可以进行混合选择操作，在运算符“.”之后列出一个向量中需要的各个分量的名称，就可以选择并重新排列这些分量。下面的代码片段说明了这个问题。

```
vec4 color= vec4(0.7,0.1,0.5,1.0);      //声明一个vec4类型的向量color
vec3 temp=color.agb;                    //相当于拿到一个向量(1.0,0.1,0.5)赋值给temp
vec4 tempL=color.aabb;                  //相当于拿到一个向量(1.0,1.0,0.5,0.5)赋值给tempL
vec3 tempLL;                            //声明了一个3维向量tempLL
tempLL.grb=color.aab;                   //对向量tempLL的3个分量赋值
```

- 从上述代码片段中可以看出，一次混合最多只能列出4个分量名称，且一次出现的各部分的分量名称必须是来自同一名称组。3个名称组分别为：xyzw、rgba、stpq，这在前面的4.2.1小节中已经介绍过。如“color.xa”就是错误的，因为分量名称没有来自同一个组。
- 各分量的名称在进行混合时可以改变顺序以进行重新排列。
- 以赋值表达式中的“=”为界，其左侧称之为L值（要写入的表达式），右侧称之为R值（所读取的表达式）。进行混合时，R值可以使用一个向量的各个分量任意地组合以及重复，而L值则不能有任何的重复分量，但可以改变分量顺序。

3．算术运算符

自加以及自减运算符“++与--”执行的操作与C语言中相同，即可以用于整数也可以用于浮点数。若在向量以及矩阵中使用，则向量或矩阵的每个元素都加1或者减1。

对标量而言，加减乘除运算与C语言基本没有区别。但若是对矩阵运算，则进行的是线性代数中的相关运算。如在矩阵上使用乘法时，执行的不再是简单的算术运算，而是线性代数的乘法。下面的代码片段说明了这些特殊的情况。

```
vec3 va=vec3(0.5,0.5,0.5);          //声明了一个 vec3 向量 va
vec3 vb=vec3(2.0,1.0,4.0);          //声明了一个 vec3 向量 vb
vec3 vc=va*vb;                      //两个向量执行按分量的乘法，加减与之类似
mat3 ma=mat3(1,2,3,4,5,6,7,8,9);    //声明了一个 mat3 矩阵 ma
mat3 mb=mat3(9,8,7,6,5,4,3,2,1);    //声明了一个 mat3 矩阵 mb
vec3 vd=va*ma;                      //执行向量与矩阵的乘法，满足线性代数的定义
mat3 mc=ma*mb;                      //执行矩阵乘法，满足线性代数的定义
```

- 向量用算术运算符运算时，执行的是各分量的算术运算。如将两个向量用“+”相加，实际执行的是向量的各分量相加得到一个新的向量。
- 关于向量与矩阵以及矩阵与矩阵的乘法都是执行的满足线性代数定义的运算。

4．其他运算符

- 关系运算符（<、>、<=、>=）只能用在浮点数或整数标量的操作中，通过关系运算符的运算将产生一个布尔型的值。
- 等于运算符（==、！=）可以用在除数组以外任何类型数据的操作中，在等于操作中将对左右两个操作数的每一个分量分别进行比较，然后得出一个布尔型的值，说明左右两个操作数是否完全相等。
- 逻辑运算符包括与（&&）、或（||）、非（!）以及异或（^^）4种操作类型，这些操作只可以用在类型为布尔标量的表达式中，不可以用在矩阵中。
- 选择运算符（？：）的使用方法也与C相同，可以用在除数组之外的任何类型中。但是要注意，第2个以及第3个表达式必须是相同

的类型。通过计算第一个逻辑表达式得到一个布尔类型的值，若为true则只计算第2个表达式，若为false则只计算第3个表达式。

• 赋值运算符中最常用的“=”在操作时，要求符号两边的操作数必须类型完全相同。这一点很特殊，与C/C++以及Java等通用编程语言不同，着色语言的赋值没有自动类型转换或提升功能。例如“float a=1；”就是错的，因为左侧的a是浮点型，右侧的1是整型。

4.2.4 类型转换

OpenGL ES着色语言没有提供类型的自动提升功能，并且对类型的匹配要求十分严格。例如 前面介绍过的，赋值表达式中的两个操作数类型必须完全相同，另外调用函数时的形参以及实参的类型也必须完全相同。

同时OpenGL ES着色语言也没有提供数据类型的强制转换功能，只能使用构造函数来完成类型转换，下面的代码片段说明了这个问题。

```
1 float f=1.0;           //声明一个浮点数 f 并赋值
2 bool b=bool(f);        //将浮点数转换成布尔类型，该构造函数将非 0 的数字转为 true，0 转为 false
3 float f1=float(b);     //将布尔值转变为浮点数，true 转换为 1.0，false 转换为 0.0
```

虽然OpenGL ES着色语言这样的类型转换设计相比其他高级语言来说不是很方便，但是却可以避免某些类型转换带来的性能、复杂性缺陷，简化了语言对应的硬件实现，可以说是利大于弊的。由于着色语言中并不存在类型转换功能，因此一些Java中熟悉的写法在着色语言中就不能使用。如：

```
float f0=1;              //声明一个浮点数并赋值
```

说明

着色语言中如此声明浮点数，会产生编译错误。这也是一个着色器编程的初学者常犯的错误，读者要多注意。

4.2.5　限定符

与其他编程语言一样，着色器中对变量也有很多可选的限定符。这些限定符中大部分只能用来修饰全局变量，主要的如表4-5所列。

表4-5　4种限定符及说明

限定符	说　明
attribute	一般用于每个顶点都各不相同的量，如顶点位置、颜色等
uniform	一般用于对同一组顶点组成的单个3D物体中所有顶点都相同的量，如当前的光源位置
varying	用于从顶点着色器传递到片元着色器的量
const	用于声明常量

下面给出了使用上述4种限定符的代码片段。

```
uniform mat4 uMVPMatrix;            //声明一个用uniform修饰的mat4类型的矩阵
attribute vec3 aPosition;           //声明一个用attribute修饰的vec3类型的向量
varying vec4 aaColor;               //声明一个用varying修饰的vec4类型的向量
const int lightsCount = 4;          //声明一个用const修饰的int类型的常量
```

限定符在使用时应该放在变量类型之前，且使用attribute、uniform以及varying限定符修饰的变量必须为全局变量。同时要注意的是，着色语言中没有默认限定符的概念，因此如果有需要，必须为全局变量明确指定需要的限定符。

下面对几种限定符进行更为详细的介绍，具体内容如下所列。

1．attribute限定符

attribute限定符顾名思义为属性限定符，其修饰的变量用来接收渲染管线传递进顶点着色器的 当前待处理顶点的各种属性值。这些属

性值每个顶点各自拥有独立的副本，用于描述顶点的各项特征，如顶点坐标、法向量、颜色、纹理坐标等。

用attribute限定符修饰的变量其值是由宿主程序（本书中为Java）批量传入渲染管线的，管线进行基本处理后再传递给顶点着色器（参考图3-21）。数据中有多少个顶点，管线就调用多少次顶点着色器，每次将一个顶点的各种属性数据传递给顶点着色器中对应的attribute变量。因此，顶点着色器每次执行将完成对一个顶点各项属性数据的处理。

从上面的介绍中可以看出，attribute限定符只能用于顶点着色器中，不能在片元着色器中使用。且attribute限定符只能用来修饰浮点数标量、浮点数向量以及矩阵变量，不能用来修饰其他类型的变量。下面的代码片段给出了在顶点着色器中正确使用attribute限定符的情况。

```
attribute vec3 aPosition;                    //顶点位置
attribute vec3 aNormal;                      //顶点法向量
```

> **提示**
>
> 从上述介绍中可以看出，若需要渲染的3D物体中有很多顶点，顶点着色器就需要执行很多次，这是很耗费时间的。另外，由于顶点着色器每次执行仅处理一个独立顶点的相关数据，可见顶点着色器的多次执行之间并没有什么逻辑依赖。因此，当今主流的GPU中都配置了不止一套顶点着色器的硬件，数量从几套到几百套不等。通过这些顶点着色器的并发执行，可以大大提高渲染速度。

前面已经提过，对于用attribute限定符修饰的变量其值是由宿主程序批量传入渲染管线的，相关代码如下。

```
1 int maPositionHandle;                           //声明顶点位置属性引用
2 maPositionHandle = GLES20.glGetAttribLocation(  //获取顶点位置属性引用的值
3 mProgram,                  //采用的着色器程序 id
4 "aPosition"                //着色器中对应的属性变量名称
5 );
6 GLES20.glVertexAttribPointer( //将顶点位置数据传送进渲染管线
7     maPositionHandle,      //顶点位置属性引用
8        3,                  //每顶点一组的数据个数(这里是 X、Y、Z 坐标因此为 3)
9        GLES20.GL_FLOAT,    //数据类型
10           false,          //是否规格化
11            3*4,           //每组数据的尺寸，这里每组 3 个浮点数值(X、Y、Z 坐标),
12                           //每个浮点数 4 个字节,共 3*4=12 个字节
13           mVertexBuffer   //存放了数据的缓冲
14        );
15    GLES20.glEnableVertexAttribArray(maPositionHandle); //启用顶点位置数据
```

- 第1行声明了一个整型变量maPositionHandle，其用来存储从着色器中获取的对应属性变量的引用值。
- 第2-5行通过glGetAttribLocation方法获取了着色器中指定名称属性变量的引用值。
- 第6-14行将已经放入缓冲中的数据批量传入渲染管线，以备管线经过基本处理后将对应的值传递给顶点着色器中的属性变量。
- 上述代码中对应于属性变量aPosition，每个顶点的数据个数为3，占用的空间为12字节，类型为GL_FLOAT。因此对于属性变量aPosition，当管线将数据传递给其时，会每次从对应的缓冲中取出12个字节，拆分成3个4字节浮点数。而着色器中名称为aPosition的属性变量类型为vec3，正好与之匹配。
- 将缓冲中的数据传递进渲染管线时需要根据对应属性变量的类型给glVertexAttribPointer方法设置合理的参数值。本书中几乎每个案例都要用到此功能，读者可以在学习中慢慢体会。

- 第15行通过属性变量引用maPositionHandle启用了批量传入管线的数据。

> **提示**
>
> 从上述代码中可以看出，主要的工作分为3步：获取对应属性变量的引用、通过引用将缓冲中的数据传入管线、启用传入的数据。上述代码片段中的工作往往不是在一个方法中完成的，具体位置情况可以参考上一章Sample3_1中的Triangle类。

从上述代码片段中可以看出，在传送属性变量的值之前，首先需要按照顶点的次序依次将与此属性变量对应的值送入缓冲，这项工作对应的代码片段如下。

```
float vertices[]=new float[] { //首先将顶点此项属性数据依次放入数组，这里是顶点坐标
  -4*UNIT_SIZE,0,               //第1个顶点的X、Y、Z坐标值
  0,0,-4*UNIT_SIZE,             //第2个顶点的X、Y、Z坐标值
   0,4*UNIT_SIZE,0,0            //第3个顶点的X、Y、Z坐标值
};
ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4); //开辟对应容量的缓冲
vbb.order(ByteOrder.nativeOrder());          //设置字节顺序为本地操作系统顺序
mVertexBuffer = vbb.asFloatBuffer();         //浮点(Float)型缓冲
mVertexBuffer.put(vertices);                 //将数组中的顶点数据送入缓冲
   mVertexBuffer.position(0);                //设置缓冲起始位置
```

> **说明**
>
> 上述代码给出了准备顶点属性变量数据缓冲的基本套路，首先将需要的数据依次放入数组，然后开辟对应容量的缓冲，最后将数组中的数据存入缓冲即可。随具体情况的变化，数据的数量、类型会有所不同，读者可以从后面章节的案例中逐步学习到。

2．uniform限定符

uniform为一致变量限定符，一致变量指的是对于同一组顶点组成的单个3D物体中所有顶点都相同的量。uniform变量可以用在顶点着色器或片元着色器中，其支持用来修饰所有的基本数据类型。与属性变量类似，一致变量的值也是从宿主程序传入的。

下面的代码片段给出了在顶点或片元着色器中正确使用uniform限定符的情况。

```
uniform mat4 uMVPMatrix;                    //总变换矩阵
uniform mat4 uMMatrix;                      //变换矩阵
uniform vec3 uLightLocation;                //光源位置
uniform vec3 uCamera;                       //摄像机位置
```

将一致变量的值从宿主程序（本书中为Java）传入渲染管线的相关代码如下。

```
int muMVPMatrixHandle;    //总变换矩阵一致变量引用
//获取着色器程序中总变换矩阵一致变量的引用
muMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
//通过一致变量引用将一致变量值传入渲染管线
GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, false, Triangle.getFianlMatrix(mMMatrix), 0);
```

说明

从上述代码中可以看出，将一致变量的值送入渲染管线比较简单，主要包括两个步骤：获取着色器程序中一致变量的引用以及调用glUniformMatrix4fv方法将一致变量的值送入渲染管线。

需要注意的是，随一致变量类型的不同将值传入渲染管线的方法也有所不同，这些方法的名称都以glUniform开头，常用的如下所列。

- glUniformN f/glUniformN fv方法，将N个浮点数传入管线，以备管线传递给由N个浮点数组成的一致变量，N的取值为1、2、3或4。
- glUniformN i/glUniformN iv方法，将N个整数传入管线，以备管线传递给由N个整数组成的一致变量，N的取值为1、2、3或4。
- glUniformMatrixN iv方法，将N×N的矩阵传入管线，以备管线传递给N×N矩阵类型的一致变量，N的取值为2、3或4。

> **提示**
>
> 上述系列方法名称中的“N ”在使用时需要根据实际情况用合法的整数代替，更详细的情况请读者参考Android SDK中附带的API文档，这里不再赘述。

3．varying限定符

要想将顶点着色器中的信息传入到片元着色器中，则必须使用varying限定符。用varying限定符修饰的全局变量又称为易变变量，易变变量可以看成是顶点着色器以及片元着色器之间的动态接口，方便顶点着色器与片元着色器之间信息的传递。

图4-1给出了易变变量的工作原理。

从图4-1中可以看出，首先顶点着色器在每个顶点中都对易变变量vPosition进行了赋值。接着在片元着色器中接收易变变量vPosition的值时得到的并不是某个顶点赋的特定值，而是根据片 元所在的位置及图元中各个顶点的位置进行插值计算产生的值。

如图4-1中顶点1、2、3的vPosition值分别为vec3（0，7，0）、vec3（5，0，0）、vec3（-5，0，0），则插值后片元a的vPosition值为vec3（1.45,2.06,0），这个值是根据3个顶点对应的着色器给vPosition赋的值、3个顶点的位置及此片元的位置由管线插值计算得到的。

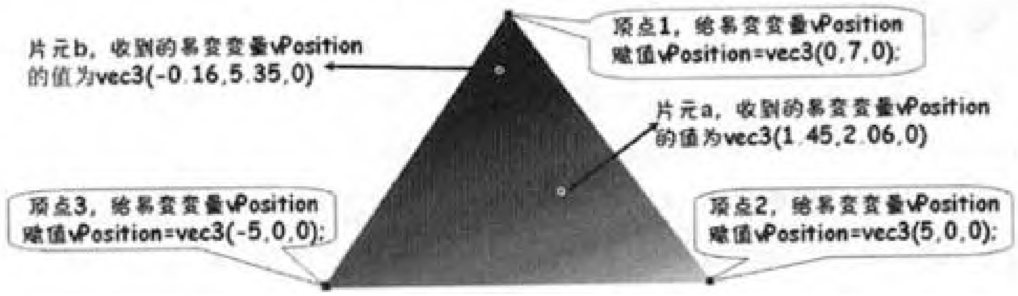


图4-1　易变变量工作原理

从上述介绍中可以看出，光栅化后产生了多少个片元，就会插值计算出多少套易变变量。同时，渲染管线就会调用多少次片元着色器。可以看出，一般情况下对一个3D物体的渲染中，片元着色器执行的次数会大大超过顶点着色器。因此GPU硬件中配置的片元着色器硬件数量往往多于顶点着色器硬件数量，通过这些硬件单元的并行执行，提高渲染速度。

提示

从上述介绍中可以看出，这就是为什么GPU在图形处理性能上远远超过同时代、同档次CPU的原因，CPU中没有这么多的并行硬件处理单元。

在顶点着色器中声明varying易变变量格式如下。

```
varying vec4 ambient;                    //环境光易变变量
varying vec4 diffuse;                    //散射光易变变量
varying vec4 specular;                   //镜面反射光易变变量
```

提示

对于顶点着色器而言，一般是既声明易变变量，又对易变变量进行赋值以传递给片元着色器。而片元着色器中声明易变变量用于

接收顶点着色器传过来的值即可，是不可以对易变变量赋值的。这一点对本书中所有的案例皆是如此，读者可以慢慢体会。

4．const限定符

用const限定符修饰的变量其值是不可以变的，也就是常量，又称为编译时常量。编译时常量在声明的时候必须进行初始化，同时这些常量在着色器外部是完全不可见的。下面的代码片段给出了如何在着色器中通过const限定符声明常量。

```
const int tempx=1;
```

提示

用const限定符修饰的变量在编译时，编译器是不需要向其分配任何运行时资源的，恰当采用可以一定程度上提高设备的运行效率，节约资源。

4.2.6　流程控制

前面已经介绍了OpenGL ES着色语言中的数据类型、对数据类型的操作以及限定符的使用，而在实际开发中，这是远远不够的。还需要进行流程控制才能写出具有完整功能的程序，本小节将介绍着色语言中流程控制相关的内容。

OpenGL ES着色语言提供了3种流程控制方式，分别由if-else条件语句、while（do-while）循环语句与for循环语句实现。下面就简单介绍3种流程控制方式的使用，具体情况如下所列。

1．if-else条件语句

该流程控制方式的基本语法有两种，如下所列。

- if（<表达式>）{语句序列}。

下面的代码片段说明了这种流程控制方式的使用。

```
int tempx=1;
if(tempx==0){
    //执行处理逻辑
}
```

• if（<表达式>）{返回为true时执行的语句序列}else {返回为false时执行的语句序列}。

下面的代码片段说明了这种流程控制方式的使用。

```
int tempx=1;
if(tempx==0){
    //执行处理逻辑
}else{
    //执行处理逻辑
}
```

> **提示**
>
> 需要注意的是，虽然很像C语言，但“<表达式>”的返回值必须为布尔类型的标量，而不能像C语言中那样随意地使用浮点数或整数。

2．while/do-while循环

开发中经常需要在满足某些条件时重复执行特定的代码，此时就可以使用while或do-while循环，具体情况如下所列。

• while循环。

while循环语句的基本语法为：while（<条件表达式>）{语句序列}。

下面的代码片段说明了while循环的使用。

```
int tempx=1;
while(tempx>=0)
{
    //执行处理逻辑
}
```

说明

while循环适用于不知道代码需要被重复执行多少次，但却有明确的终止条件的循环流程。只要条件表达式判断为true，后面的语句序列就将重复执行，直到条件表达式为false为止。对于while循环而言，如果第一次判断条件表达式就返回false，则循环一次都不执行。

- do-while循环。

do-while循环语句的基本语法为：**do{语句序列} while（<条件表达式>）。**

下面的代码片段说明了do-while循环的使用。

```
int tempx=1;
do{
    //执行处理逻辑
}while(tempx<=0);
```

说明

do-while循环与while循环基本相同，其先执行语句序列再判断条件表达式，直到表达式返回为false。这样的话do后面的语句序列将至少执行一次，这是其与while循环最大的不同点。

3．for循环

若明确知道循环的执行次数，则应该使用for循环，for循环语句的基本语法为：

```
for(初始化表达式;条件表达式;更新语句列表) {语句序列}
```

- 初始化表达式用来声明并初始化一个或者多个相同类型的变量，一般用以控制循环的次数。
- 条件表达式则只能有一个，且条件表达式的返回值必须为布尔类型。若不写条件表达式，则相当于永远为true。

- 更新语句列表在每次循环之后才执行，一般用于改变循环控制变量的值。这里可以写多个语句也可以不写语句，写多个语句时语句间用“，”隔开。

下面的代码片段说明了for循环的具体使用。

```
for(int i=0;i<13;i++){
    //执行处理逻辑
}
```

> **说明**
>
> for循环首先初始化循环控制变量，然后计算条件表达式，若返回为true则执行循环体一次。执行完循环体后执行更新语句列表，然后再判断条件表达式是否为true，若为true则再次执行以上流程，若为false则结束循环。

4．break与continue循环控制

与C语言类似，在循环体中，同样可以使用break与continue跳出循环，具体情况如下所列。

- break语句在循环控制中用于中断循环，如果在循环体中执行了break语句，则循环中断并退出。要注意的是，在使用了多层嵌套的循环中，使用break语句跳出的是离其最近的一层循环。
- continue语句在循环控制中用于跳过本次进入下一次循环，如果在循环体中执行了continue语句，则本次循环结束，转而执行条件表达式，若条件表达式为true则继续执行下一次循环。

> **提示**
>
> 要特别注意的是，着色语言中没有包含C/C++中常用的switch语句，需要时可以使用if-else语句的嵌套来实现。

4.2.7　函数的声明与使用

与C语言中相同，着色语言中也可以开发自定义的函数，基本语法如下。

```
<返回类型> 函数名称 ([<参数序列>]){ /*函数体*/}
```

从上述语法中可以看出，声明函数时要包含4个组成部分，分别是“返回类型”、“函数名称”、“参数序列”、“函数体”。

- 返回类型根据需要可以是前面4.2.1节中介绍的除采样器之外的任何类型。
- 函数名称要满足着色语言的命名规定，可以包含字母、数字、下划线，其中起始字符不能为数字，可以为字母或下划线。
- 参数序列放在一对圆括号中，若没有则为空。
- 函数体包含在一对花括号中，包含完成函数功能所需要的语句。

另外，参数序列中的参数除了可以指定类型外，还可以指定用途。具体方法为用参数用途修饰符进行修饰，常用的参数用途修饰符如下所列。

- “in”修饰符，用其修饰的参数为输入参数，仅供函数接收外界传入的值。若某个参数没有明确给出用途修饰符，则等同于使用了“in”修饰符。
- “out”修饰符，用其修饰的参数为输出参数，在函数体中对输出参数赋值可以将值传递到调用其的外界变量中。对于输出参数，要注意的是在调用时不可以使用字面常量。
- “inout”修饰符，用其修饰的参数为输入输出参数，具有输入与输出两种参数的功能。

> **提示**
>
> 从上述用途修饰符的讲解中可以看出，着色语言中函数返回信息的渠道除了返回值外还有输出参数，在需要的时候恰当使用可以增加开发的灵活性。

4.2.8 片元着色器中浮点变量精度的指定

片元着色器中使用浮点相关类型的变量时与顶点着色器中有所不同，在顶点着色器中直接声明使用即可，而在片元着色器中必须指定精度，若不指定精度可能会引起编译错误。指定精度的方法如下面的代码片段所示。

```
lowp float color;                //指定名称为color的float型变量精度为lowp
varying mediump vec2 Coord;      //指定名称为Coord的vec2型变量精度为mediump
highp mat4 m;                    //指定名称为m的mat4型变量精度为highp
```

从上述代码片段中可以看出，精度有3种选择，lowp、mediump及highp。这3种选择分别代表低、中、高3种精度等级，在不同的硬件中实现可能会有所不同。一般情况下，使用mediump即可。另外，还可以看出所谓浮点相关类型不单包括标量类型float，还包括与之对应的向量类型vec2、vec3、vec4，以及与之对应的矩阵类型mat2、mat3、mat4。

如果在开发中同一个片元着色器中浮点相关类型的变量都选用同一种精度，则可以指定整个着色器中浮点相关类型的默认精度，语法如下。

```
precision <精度> <类型>;
```

- 精度可以选择lowp、mediump及highp 3种之一。
- 类型一般为float，这不单表示为浮点标量类型float指定了精度，还表示对浮点类型相关的 向量、矩阵也指定了默认精度。因此一般开发中经常将片元着色器的第一句写为“precision mediump float;”，本书中的大部分案例都是如此。

提示

类型不但可以是float，还可以是int或任何的采样器类型。由于整型相关类型和采样器类型并不一定要求指定精度，因此用的不多。

4.2.9 程序的基本结构

前面几个小节介绍了着色语言很多方面独立的基本知识，本小节将介绍着色器程序的基本结构。一个着色器程序一般由3大部分组成，主要包括：全局变量声明、自定义函数、main函数。下面的代码片段给出了一个完整的顶点着色器程序。

```
1uniform mat4 uMVPMatrix;          //总变换矩阵
2attribute vec3 aPosition;         //顶点位置
3attribute vec2 aTexCoor;          //顶点纹理坐标
4varying vec2 vTextureCoord;       //用于传递给片元着色器的变量
5void positionShift()              //根据总变换矩阵计算此次绘制此顶点位置的方法
6{
7    gl_Position = uMVPMatrix * vec4(aPosition,1);
8}
9void main()                       //主函数
10   {
11     positionShift();            //根据总变换矩阵计算此次绘制此顶点位置
12     vTextureCoord = aTexCoor;   //将接收的纹理坐标传递给片元着色器
13   }
```

- 第1-4行为全局变量的声明，根据具体情况的不同可能会有增加或减少。
- 第5-8行为自定义的函数，这一部分根据需要可能没有，也可能有很多不同的函数。
- 第7行的gl_Position是顶点着色器中的内建变量，这部分内容下一节将进行详细的介绍。
- 第9-13行为主函数main，这是每个着色器里面都必须有的部分。

提示

与很多高级语言不同，着色器程序中要求被调用的函数必须在调用之前声明。就如上述代码中首先在第5行声明了positionShift函数，然后才能在第11行进行调用。这也是初学者易犯的一个错误，读者在开发中要多留心。

4.3　特殊的内建变量

着色器代码的开发中会用到很多变量，其中大部分可能是由开发人员根据需求自定义的，但着色器中也提供了一些用来满足特定需求的内建变量。这些内建变量不需要声明即可使用，一般用来实现渲染管线固定功能部分与自定义顶点或片元着色器之间的信息交互。

内建变量根据信息传递的方向可以分为两类，输入与输出变量。输入变量负责将渲染管线中固定功能部分产生的信息传递进着色器，输出变量负责将着色器产生的信息传递给渲染管线中固定功 能部分。

4.3.1　顶点着色器中的内建变量

顶点着色器中的内建变量主要是输出变量，包括gl_Position、gl_PointSize等。在顶点着色器中应该根据需要给这些内建变量赋值，以便由渲染管线中的图元装配与光栅化等后续固定功能阶段进行进一步的处理。

下面对顶点着色器中的几个内建输出变量进行详细介绍，具体如下所列。

- gl_Position。

顶点着色器从应用程序中获得原始的顶点位置数据，这些原始的顶点数据在顶点着色器中经过平移、旋转、缩放等数学变换后，生成新的顶点位置。新的顶点位置通过在顶点着色器中写入gl_Position传递到渲染管线的后继阶段继续处理。

gl_Position的类型是vec4，写入的顶点位置数据也必须与其类型一致。几乎在所有的顶点着色器中都必须对gl_Position写入适当的值，否则后继阶段的处理结果将是不确定的。

- gl_PointSize。

顶点着色器中可以计算一个点的大小（单位为像素），并将其赋值给gl_PointSize（标量float类型）以传递给渲染管线。如果没有明

确赋值的话，就是采用默认值1了。gl_PointSize的值一般只有在采用了点绘制方式之后才有意义，关于绘制方式的问题在后面的章节会进行详细的介绍。

4.3.2　片元着色器中的内建变量

片元着色器中的内建变量分为输入变量以及输出变量两种，本小节将对这两种不同的内建变量分别进行详细介绍。

1．内建输入变量

片元着色器中的内建输入变量主要有gl_FragCoord以及gl_FrontFacing。这两个内建变量都是只读的，由渲染管线中片元着色器之前的阶段生成，其具体含义如下。

- gl_FragCoord。

内建变量gl_FragCoord（vec4类型）中含有当前片元相对于窗口位置的坐标值x、y、z与1/w（如图4-2所示）。其中x与y分别为片元相对于窗口的二维坐标。如果窗口的大小为800*480（单位为像素），那么x的取值范围为0～800，y的取值范围为0～480，z部分为该片元的深度值。

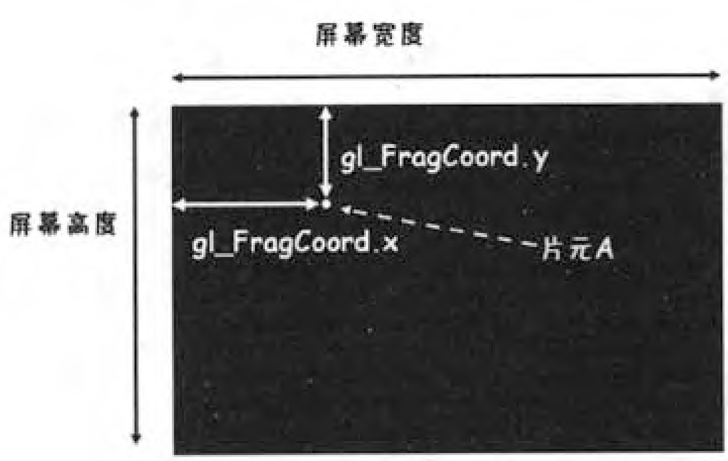


图4-2　gl_FragCoord包含的窗口坐标信息

提示

通过该内建变量可以实现与窗口位置相关的操作，例如仅绘制窗口中指定区域的内容等。

- gl_FrontFacing。

gl_FrontFacing是一个布尔型的内建变量，通过读取该内建变量的值可以判断正在处理的片元 是否属于在光栅化阶段生成此片元的对应图元的正面。如果属于正面，gl_FrontFacing的值为true，反之为false。其一般用于开发双面光照功能相关的应用程序中。

对于点、线段等没有正反面之分的图元，其生成的片元都会被默认为是正面的。对于三角形图元来说，其正反面取决于应用程序中对卷绕的设置及图元中顶点的具体卷绕情况。

提示

关于卷绕的问题，在后面的章节中会进行详细介绍，这里读者有个简单概念即可。

2．内建输出变量

片元着色器中的内建输出变量主要有gl_FragColor与gl_FragData，在片元着色器中根据具体情况需要给这两个内建变量写入值。下面列出了这两个内建变量的具体情况。

- gl_FragColor。

gl_FragColor（vec4类型）内建变量用来由片元着色器写入计算完成的片元颜色值，此颜色值将送入渲染管线的后继阶段进行处理。

- gl_FragData。

gl_FragData内建变量本身是一个vec4类型的数组，写入时要给出下标，如“gl_FragData[0]”。通过其写入的信息将供渲染管线中的后继过程使用。

提示

实际开发中，对上述两个内建输出变量赋值时应该根据具体情况仅选用其中的一个，而不应该同时对两个都进行赋值。若执行了

discard操作（后面章节介绍）则两个内建变量都不需要写入值了。就笔者开发的经验而言，大部分情况下都是给gl_FragColor内建变量赋值的，读者在后面大量的例子中也会逐渐体会到。

4.4　着色语言的内置函数

与其他高级语言类似，为了方便开发，OpenGL ES着色语言中也提供了很多的内置函数。这些函数大都已经被重载，一般具有4种变体，分别用来接收和返回float、vec2、vec3以及vec4类型的值。

这些内置函数通常是以最优方式来实现的，有部分函数甚至由硬件直接支持，这大大地提高了执行效率。大部分内置函数同时适用于顶点着色器与片元着色器，但是也有部分内置函数只适用于顶点着色器或者片元着色器。内置函数按照设计目的可以分为3个类别。

- 提供独特硬件功能的访问接口，如纹理采样系列的函数，这些函数用户是无法自己开发的。着色语言通过提供特定内置函数对这些硬件功能进行封装，建立了用户调用这些硬件功能的接口。
- 简单的数学函数，如abs（求模）、floor（取整）等。这些数学函数本身非常简单，开发人员也可以自己开发，但可能由于对底层硬件的不了解，采用的实现方式很低效。而内置函数是厂商根据硬件的特点用最高效的方式实现的，调用内置函数来完成这些简单的操作不但可以提高开发效率还可以提高执行效率。
- 一些复杂的函数，如三角函数等，用户可以自己编写，但是编写过程特别烦琐，要用到很 多高等数学的知识。不但开发烦琐，可以想象效率也会很低下。而当下的主流硬件往往都有进行这些计算的指令，因此对这些操作也提供了高效的内置函数。

4.4.1　角度转换与三角函数

角度转换与三角函数同时适用于顶点着色器与片元着色器，并且每个角度转换与三角函数都有4种重载变体，具体情况如表4-6所列。

表4-6　角度转换与三角函数

内置函数签名	说明
genType radians (genType degrees)	此函数功能为将角度转换为弧度，即返回值results-（π/180）* degrees degrees参数表示需要转换的角度
genType degrees (genType radians)	此函数功能为将弧度转换为角度，即返回值result=（180/π）* radians radians参数表示需要转换的弧度
genType sin (genType angle)	此函数为标准的正弦函数，其返回值范围是[-1，1] radians为正弦函数的参数，以弧度为单位
genType cos (genType angle)	此函数为标准的余弦函数，其返回值范围是[-1,1] radians为余弦函数的参数，以弧度为单位
genType tan (genType angle)	此函数为标准的正切函数，radians为正切函数的参数，以弧度为单位
genType asin (genType x)	此函数为标准的反正弦函数，其返回值范围是[- π /2, π /2] x为反正弦函数的参数，其取值范围是[-1,1]如果x的绝对值大于1，那么结果不确定
genType acos (genType x)	此函数为标准的反余弦函数，其返回值范围是[0， π] x为反余弦函数的参数，其取值范围是[-1,1]如果x的绝对值大于1,那么结果不确定
genType atan (genType y,	此函数为标准的反正切函数，其返回值范围是[-π， π] x与y为反正切函数的参数，而实际传入反正切函数的是y/x的值，其中通过x与y的符号用来确定角度所在的象限。如果x与y的值全为零，那么返回值不确定

genType x)	
genType atan (genType y_over_x)	此函数为反正切函数，其返回值范围是[-π/2, π/2] y_over_x为反正切函数的参数，不存在范围限制

说明

表4-6中genType代表的数据类型有float、vec2、vec3以及vec4。其中float指的是浮点数标量，vec2指的是二维的浮点数向量，vec3指的是三维的浮点数向量，vec4指的是四维的浮点数向量。有关数据类型的详细内容，请读者参考本章4.2.1小节。

由于sin（正弦函数）与cos（余弦函数）是周期性平滑变化的函数（如图4-3所示），因此除了可以做三角函数之外，还有很多使用方式。例如，可以用来模拟水波纹效果，只需要选择其中的一个函数，然后做周期性变换即可，后面的章节会有这方面的案例。

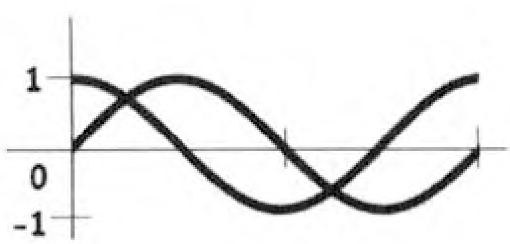


图4-3　正弦与余弦函数

4.4.2　指数函数

前一小节介绍了角度转换与三角函数，接下来在本小节中将介绍指数函数。指数函数同时适用于顶点着色器与片元着色器，具体情况如表4-7所列。

表4-7　指数函数

内置函数签名	说　明
genType pow (genType x, genType y)	此函数返回x的y次方，即x^2 x与y分别为本函数的两个参数，其中x为指数函数的底数，y为指数函数的指数。如果x值小于0，那么返回值不确定。如果x值等于零，并且y值小于等于0，那么返回值不确定
genType exp (genType x)	此函数返回e（数学常数，值近似等于2.718281828）的x次方，即e^x x为本函数的参数，代表指数
genType log (genType x)	此函数返回以e为底的x的对数，即loge（x）。也就是说，如果返回值为y,那么也满足方程$x=e^y$ 。x为本函数的参数。如果x值小于等于0,那么返回值不确定
genType exp2 (genType x)	此函数返回2的x次方，即2^x 。x为本函数的参数，不存在范围限制
genType log2 (genType x)	此函数返回以2为底的x的对数，即$\log_2$ （x）。也就是说，如果返回值为y,那么也满足方程$x=2^y$ 。x为本函数的参数。如果x值小于等于0,那么返回值不确定
genType sqrt (genType x)	此函数返回x的平方根，即 $\sqrt{x}$ x为本函数的参数，如果x值小于0，那么结果不确定
genType inversesqrt (genType x)	此函数返回x正平方根的倒数，即 $\frac{1}{\sqrt{x}}$ x为本函数的参数。如果x值小于等于0,那么结果不确定

说明

表4-7中genType代表的数据类型有float、vec2、vec3以及vec4。有关数据类型的详细内容，请读者参考本章4.2.1小节。

4.4.3 常见函数

介绍完指数函数后，接下来本小节主要介绍着色语言中的常见函数。这些函数可同时用于顶点着色器与片元着色器，具体情况如表4-8所列。

表4-8 常见函数

内置函数签名	说明
genType abs (genType x)	此函数的功能为求绝对值。x为本函数的参数，如果x≥0，那么返回值为x，如果x<0，那么返回值为-x

续表

内置函数签名	说　明
genType sign (genType x)	此函数的功能是与 0 进行比较，进而返回相应的值。x 为本函数的参数：如果 x>0，则返回 1.0；如果 x=0，则返回 0；如果 x<0，则返回-1.0
genType floor (genType x)	此函数功能为返回小于或者等于 x 的最大的整数值 x 为本函数的参数，不存在取值范围限制
genType ceil (genType x)	此函数功能为返回大于或者等于 x 的最小的整数值 x 为本函数的参数，不存在取值范围限制
genType fract (genType x)	此函数功能为返回 x- floor (x)的值 x 为本函数的参数，不存在取值范围限制
genType mod (genType x, float y)	此函数的功能是进行取模运算，相当于 Java 语言中的"x%y"。对于 x 中的各组成元素，使用浮点数 y，最后返回 x-y*floor(x/y)
genType mod (genType x, genType y)	此函数的功能是进行取模运算，相当于 Java 语言中的"x%y"。对于 x 中的各组成元素，使用 y 中的对应组成元素，最后返回 x-y*floor(x/y)
genType min (genType x, genType y) genType min (genType x, float y)	此函数的功能是获得最小值。x 与 y 为本函数的两个参数，如果 x 与 y 中的组成元素满足 y<x，则返回 y，否则返回 x
genType max (genType x, genType y) genType max (genType x, float y)	此函数的功能是获得最大值。x 与 y 为本函数的两个参数，如果 x 与 y 中的组成元素满足 y<x，则返回 x，否则返回 y
genType clamp (genType x,genType minVal, genType maxVal) genType clamp (genType x,float minVal,float maxVal)	此函数主要返回 min (max (x, minVal), maxVal) x、minVal 与 maxVal 为本函数的 3 个参数，如果 minVal>maxVal，则返回值不确定
genType mix (genType x,genType y,genType a) genType mix (genType x,genType y,float a)	此函数主要功能为使用因子 a 对 x 与 y 执行线性混合，即返回 x*(1-a)+y*a
genType step (genType edge, genType x) genType step (float edge, genType x)	此函数通过 x 与 edge 比较返回相应的值。edge 与 x 为本函数的两个参数，不存在范围限制。如果 x<edge，则返回 0.0，否则返回 1.0
genType smoothstep (genType edge0, genType edge1,genType x) genType smoothstep (float edge0,float edge1,genType x)	此函数功能是通过 x 与 edge0、edge1 进行比较返回相应的值 edge0、edge1 与 x 为本函数的两个参数，不存在范围限制。如果 x<=edge0，则返回 0；如果 x>=edge1，则返回 1.0，当 edge0<x<edge1 时，本函数则返回 0 与 1 之间平滑的 Hermite 插值。关于 Hermite 插值有兴趣的读者可以查阅相关的数学资料，本书篇幅有限，不再赘述

说明

表4-8中genType代表的数据类型有float、vec2、vec3以及vec4，有关数据类型的详细内容，请读者参考本章4.2.1小节。

了解了各常见函数的基本情况后，下面再对一些常见函数进行更为详细的介绍。

- abs函数。

abs函数不会产生负值，可以用来在一个平滑函数中引入间断，此函数的图形表示如图4-4所示。

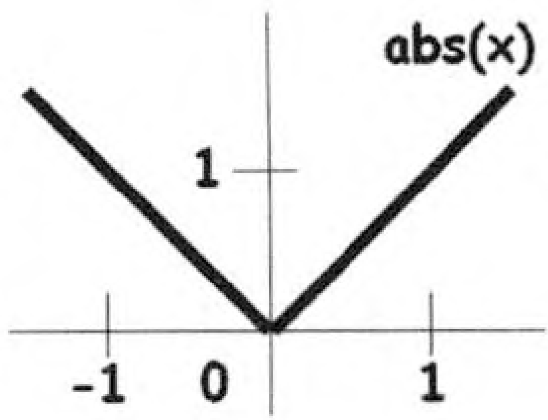


图4-4　abs函数

- sign函数。

sign函数可能返回的结果有-1、0以及1，该返回值是根据传入的参数值决定的。其为一个不连续函数，图形表示如图4-5所示。

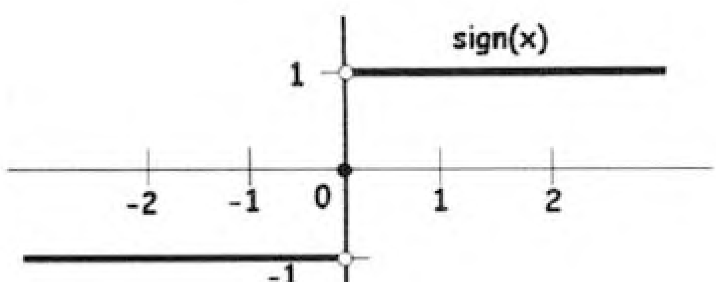


图4-5　sign函数

• floor函数。

floor函数也是一种不连续函数，图形表示如图4-6所示。此函数会根据参数值返回小于或者等于该值的最大整数。也就是将参数值的分数部分丢弃，只取整数部分。

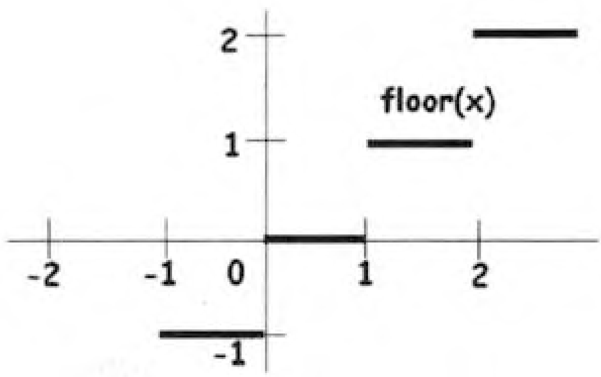


图4-6　floor函数

• ceil函数。

ceil函数与floor函数相似，也是不连续函数，只是ceil函数总是返回大于或者等于参数值的最小的整数，其图形表示如图4-7所示。

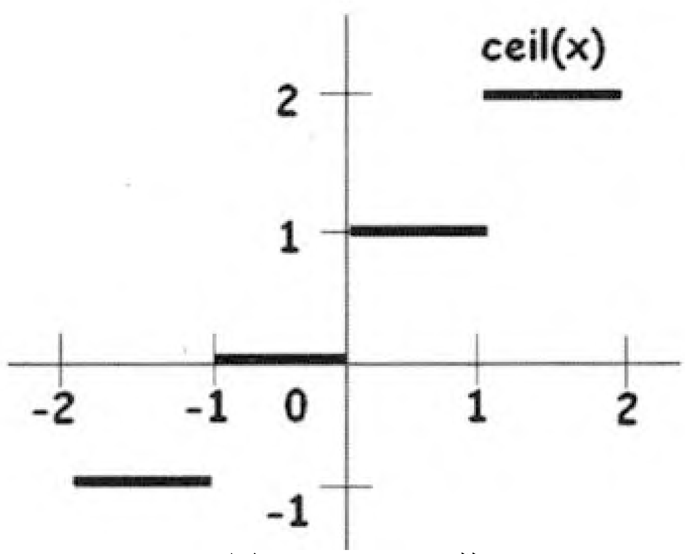


图4-7　ceil函数

说明

通过图4-6和图4-7的比较，读者可以很容易地发现，只需要将floor函数向左移动一个单位值即可获得ceil函数。

- fract函数。

fract函数也是不连续的，其中每段的斜率均为1，图形表示如图4-8所示。

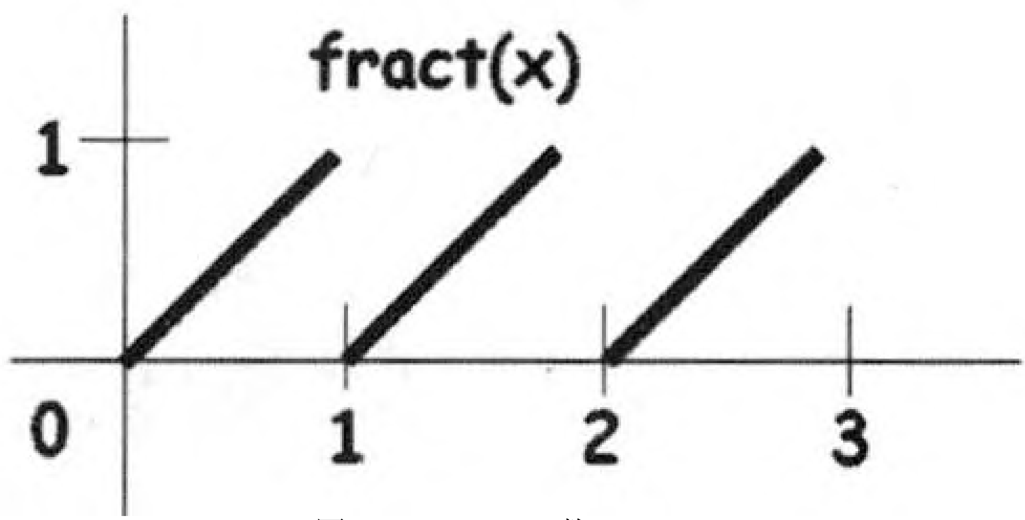


图4-8　fract函数

- min函数。

min函数的返回值是根据传入的两个参数的大小而定的。如果两个参数x与y满足y<x，那么返回y，否则返回x，其图形表示如图4-9所示。

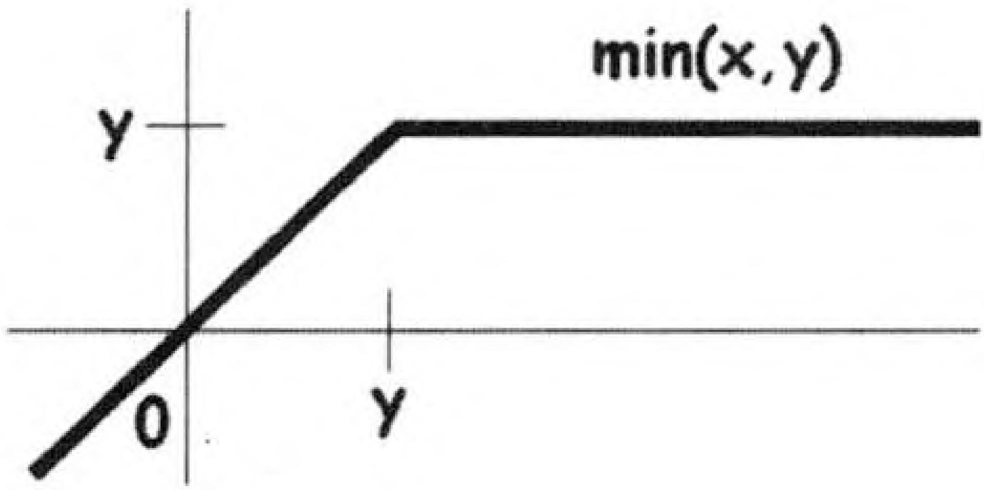


图4-9　min函数

- max函数。

max函数与min函数相似，其返回值也是根据传入的两个参数的大小而定的。如果两个参数x与y满足y<x，那么返回x，否则返回y，其图形表示如图4-10所示。

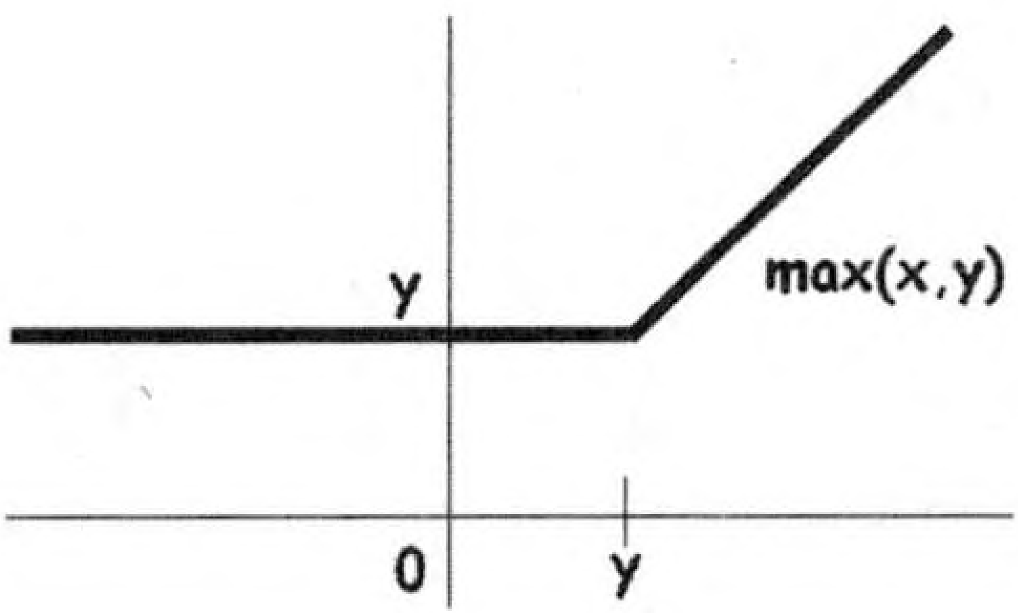


图4-10　max函数

- clamp函数。

clamp函数是连续函数，其图形表示如图4-11所示。该函数会根据传入的x值、minVal值以及maxVal值确定其返回值。其要求传入的minVal值必须小于maxVal值，否则返回值不确定。

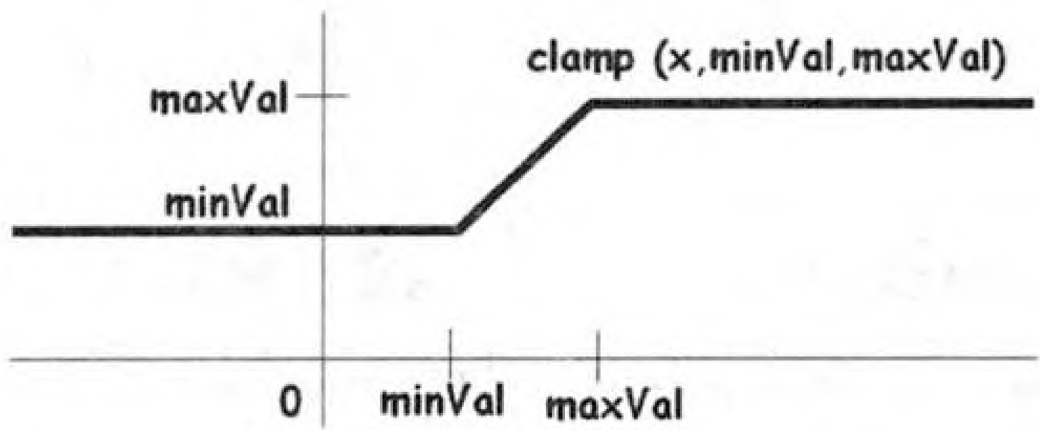


图4-11　clamp函数

- step函数。

step函数是不连续函数，其图形表示如图4-12所示。该函数主要是根据传入的参数edge与x确定返回值，如果x<edge，则返回0.0，否则返回1.0。

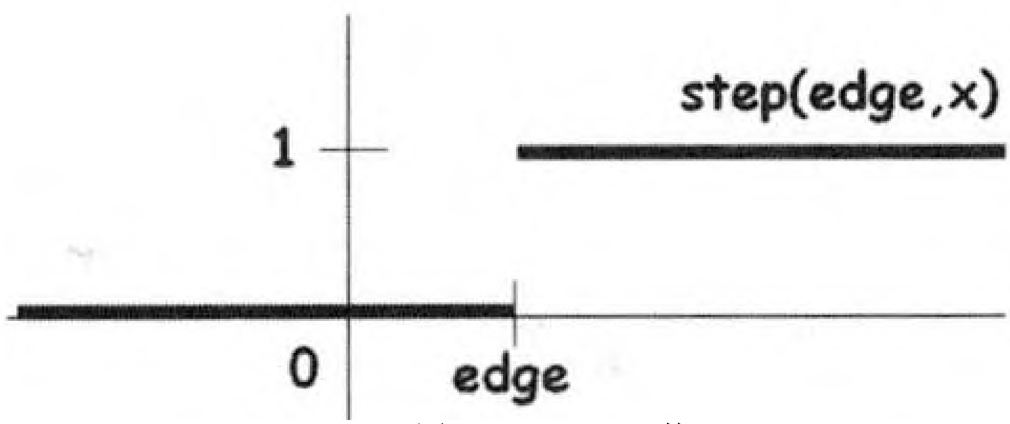


图4-12　step函数

- smoothstep函数。

smoothstep为连续函数，其图形表示如图4-13所示。此函数主要用来在两个值之间（0～1）进行平滑过渡，具体计算方法如下面的代

码片段所示。

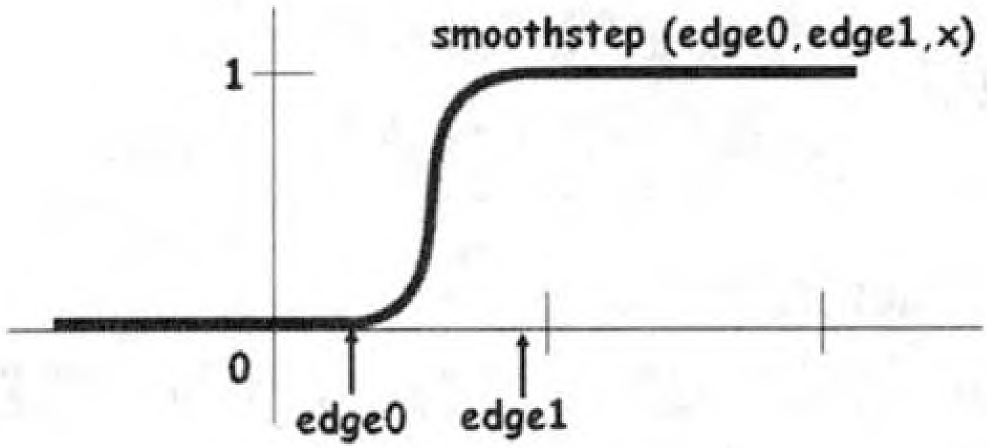


图4-13　smoothstep函数示意图

```
float t;                                          //声明变量t，用来存储平滑过渡中的值
t=clamp((x-edge0)/(edge1-edge0), 0.0, 1.0);      //计算在x位置对edge1与edge0之间的线性插值
return t*t*(3.0-2.0*t);                           //产生对应此x位置的平滑过渡值
```

提示

介绍smoothstep函数的计算方法时只是使用了浮点数标量，vec2、vec3以及vec4的情况可以依此类推，也就是对每个分量都进行相同方式的计算。

4.4.4　几何函数

前一小节介绍了常见函数，接下来在本小节中将主要介绍几何函数。几何函数同时适用于顶点着色器与片元着色器，主要用于对向量进行操作，具体情况如表4-9所列。

表4-9　几何函数

表 4-9　几何函数

内置函数签名	说　明
float length (genType x)	此函数的功能是返回向量 x 的长度，即 $\sqrt{x[0]*x[0]+x[1]*x[1]+\ldots}$ x 为本函数的参数，不存在范围限制

内置函数签名	说　明
float distance (genType p0, genType p1)	此函数的功能是返回 p0 与 p1 之间的距离，即 length(p0-p1) p0 与 p1 为本函数的参数，不存在范围限制
float dot (genType x, genType y)	此函数的功能是返回向量 x 与 y 的点积，即 x[0]*y[0]+x[1]*y[1]+x[2]*y[2]+... x 与 y 为本函数的参数，不存在范围限制
vec3 cross (vec3 x, vec3 y)	此函数的功能是返回向量 x 与 y 的叉积，即返回值为： $\begin{bmatrix} x[1]*y[2]-y[1]*x[2] \\ x[2]*y[0]-y[2]*x[0] \\ x[0]*y[1]-y[0]*x[1] \end{bmatrix}$ x 与 y 为本函数的参数，不存在范围限制
genType normalize (genType x)	此函数功能为返回与向量 x 方向相同，并且长度为 1 的向量，也就是对一个向量进行规格化。x 为本函数的参数，不存在范围限制
genType faceforward(genType N,genType I, genType Nref)	此函数功能是根据 dot(Nref,I)的值返回相应的值，如果 dot(Nref,I)<0，则返回 N，否则返回-N。N、I 与 Nref 为本函数的参数，不存在范围限制
genType reflect (genType I, genType N)	此函数功能是根据传入的入射向量 I 以及表面法向量 N，返回反射方向的向量，为了得到希望的结果，传入的表面法线向量需要先规格化
genType refract(genType I, genType N,float eta)	此函数功能是根据传入的入射向量 I、表面法向量 N 以及折射系数 eta，返回折射向量。为了得到期望的结果，传入的入射向量 I 与表面法向量 N 需要先规格化

说明

表4-9中genType代表的数据类型有float、vec2、vec3以及vec4，有关数据类型的详细内容，请读者参考本章4.2.1小节。对于只具有float参数的length、distance与normalize函数并不是非常有用，但考虑到着色语言的完整性，这些函数被定义。length（float x）返回值为|x|、distance（float p0, float p1）返回值为|p0-p1|以及normalize（float x）返回值为1。

了解了各几何函数的基本情况后，下面再对一些几何函数进行更为详细的介绍。

- dot函数。

此函数的功能是返回两个向量x与y的点积。两个向量点积的符号，与两个向量的夹角是直接相关的。夹角大于零并且小于90°，则点积所得结果大于零；夹角等于90°，则点积所得结果为0；夹角大于90°，则点积小于0；具体情况如图4-14所示。

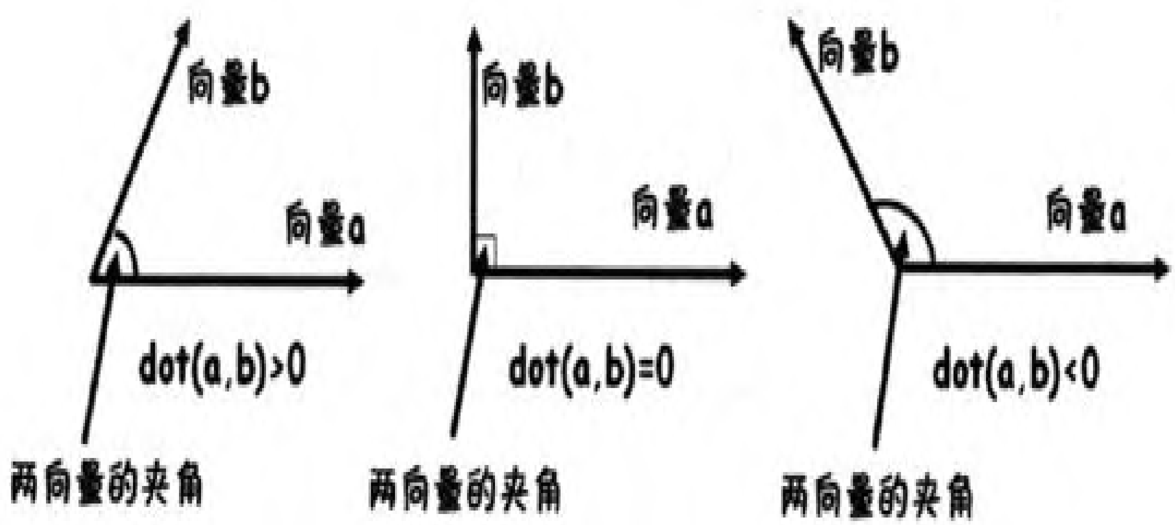


图4-14　向量点积

- cross函数。

此函数的功能是返回两个向量x与y的叉积，两个向量叉积的绝对值即为这两个向量所在四边形的面积，效果如图4-15所示。

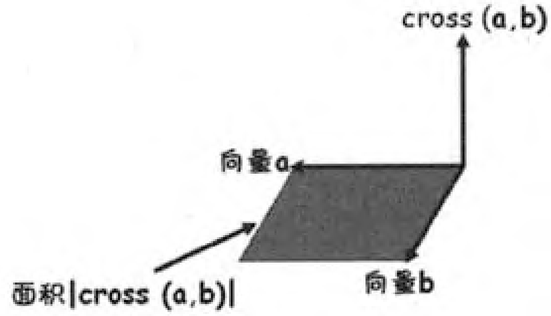


图4-15　向量叉积

- reflect函数。

此函数功能是根据传入的入射向量I以及表面法向量N，返回反射方向向量，如图4-16所示。

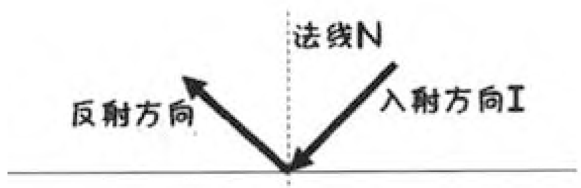


图4-16　向量反射

- refract函数。

此函数功能是根据传入的入射向量I、表面法向量N以及折射系数eta，返回折射向量，如图4-17所示。此函数的具体计算方法如下面的代码片段所示。

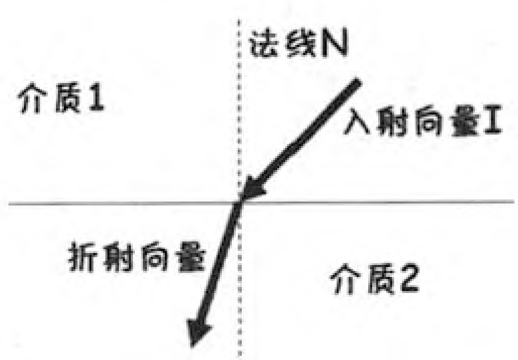


图4-17　折射效果图

```
k= 1.0 - eta * eta *(1.0 - dot(N, I) * dot(N, I))   //计算判断系数k
if (k < 0.0){
    return genType(0.0);                            //若符合全反射则无折射向量
}else{
    return eta * I - (eta *dot(N, I) + sqrt(k)) * N;} //若不符合全反射根据斯涅尔定律计算折射向量
```

> **说明**
>
> 折射系数eta与介质1与介质2的折射率有关，介质1的折射率除以介质2的折射率即为折射系数。这里只是对反射以及折射函数进行了简单介绍，后面会有具体的章节为读者介绍使用反射与折射函数实现酷炫的效果。

4.4.5　矩阵函数

矩阵函数目前只有一个，可以用于顶点着色器，也可以用于片元着色器，具体情况如表4-10所列。

表4-10　矩阵函数

内置函数签名	说　明
mat matrixCompMult (mat x, mat y)	此函数按各个部分将矩阵 x 与矩阵 y 相乘，即返回值 result[i][j]是 x[i][j]与 y[i][j]标量的乘积

> **说明**
>
> 表4-10中mat代表的数据类型有mat2、mat3以及mat4，有关数据类型的详细内容，请读者参考本章4.2.1小节。本函数实现的是矩阵的标量乘法，若希望进行线性代数中定义的矩阵乘法则需要使用乘法运算符（*）。

以三维矩阵（mat3）x、y为例，matrixCompMult（x，y）具体执行的操作如下。

$$\begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{bmatrix}\begin{bmatrix} y_{00} & y_{01} & y_{02} \\ y_{10} & y_{11} & y_{12} \\ y_{20} & y_{21} & y_{22} \end{bmatrix}=\begin{bmatrix} x_{00}y_{00} & x_{01}y_{01} & x_{02}y_{02} \\ x_{10}y_{10} & x_{11}y_{11} & x_{12}y_{12} \\ x_{20}y_{20} & x_{21}y_{21} & x_{22}y_{22} \end{bmatrix}$$

若执行的是x*y则大有不同，具体情况如下所示。

$$\begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{bmatrix}*\begin{bmatrix} y_{00} & y_{01} & y_{02} \\ y_{10} & y_{11} & y_{12} \\ y_{20} & y_{21} & y_{22} \end{bmatrix}=$$

$$\begin{bmatrix} x_{00}y_{00}+x_{01}y_{10}+x_{02}y_{20} & x_{00}y_{01}+x_{01}y_{11}+x_{02}y_{21} & x_{00}y_{02}+x_{01}y_{12}+x_{02}y_{22} \\ x_{10}y_{00}+x_{11}y_{10}+x_{12}y_{20} & x_{10}y_{01}+x_{11}y_{11}+x_{12}y_{21} & x_{10}y_{02}+x_{11}y_{12}+x_{12}y_{22} \\ x_{20}y_{00}+x_{21}y_{10}+x_{22}y_{20} & x_{20}y_{01}+x_{21}y_{11}+x_{22}y_{21} & x_{20}y_{02}+x_{21}y_{12}+x_{22}y_{22} \end{bmatrix}$$

4.4.6　向量关系函数

向量关系函数的功能为将向量的各分量进行关系比较运算（<，<=，>，>=，=，！=），生成标量的布尔值结果，具体情况如表4-11所列。

表4-11　向量相关函数

内置函数签名	说明
bvec lessThan(vec x, vec y) bvec lessThan(ivec x, ivec y)	此函数功能是返回向量x与y中的各个分量执行x<y的结果
bvec lessThanEqual(vec x, vec y) bvec lessThanEqual(ivec x, ivec y)	此函数功能是返回向量x与y中的各个分量执行x<=y的结果
bvec greaterThan(vec x, vec y) bvec greaterThan(ivec x, ivec y)	此函数功能是返回向量x与y中的各个分量执行x>y的结果
bvec greaterThanEqual(vec x, vec y) bvec greaterThanEqual(ivec x, ivec y)	此函数功能是返回向量x与y中的各个分量执行x>=y的结果
bvec equal(vec x, vec y) bvec equal(ivec x, ivec y) bvec equal(bvec x, bvec y)	此函数功能是返回向量x与y中的各个分量执行x==y的结果
bvec notEqual(vec x, vec y) bvec notEqual(ivec x, ivec y) bvec notEqual(bvec x, bvec y)	此函数功能是返回向量x与y中的各个部分执行x!=y的结果
bool any(bvec x)	如果x中任何一个分量为true，则返回true
bool all(bvec x)	x中的所有组成元素都为true，则返回true

bvec not(bvec x)	对于x的各个分量执行的逻辑非运算

说明

表4-11中vec代表的数据类型有vec2、vec3以及vec4；ivec代表的数据类型有ivec2、ivec3以及ivec4；bvec代表的数据类型有bvec2、bvec3以及bvec4。有关数据类型的详细内容，请读者参考本章4.2.1小节。

4.4.7　纹理采样函数

纹理采样函数用来根据指定的纹理坐标从采样器对应的纹理中进行采样，返回值类型都是vec4，代表采样得到的颜色值。大部分纹理采样函数既可以用于顶点着色器也可以用于片元着色器但有个别的仅适用与片元着色器，具体情况如表4-12所列。

表4-12　纹理采样函数

内置函数签名	说　明
vec4 texture2D (sampler2D sampler, vec2 coord, [float bias]) vec4 texture2DProj (sampler2D sampler, vec3 coord, [float bias]) vec4 texture2DProj (sampler2D	此系列函数功能为使用二维纹理坐标coord在由sampler参数指定的2D纹理中执行纹理采样。对于带有Proj后缀的函数，会将纹理坐标（coord.s，coord.t）除以coord的最后一个部分（coord.q）。对于类型为vec4的coord，忽略其第三部分（coord.p）

sampler, vec4 coord, [float bias]) vec4 texture2DLod (sampler2D sampler, vec2 coord, float lod) vec4 texture2DProjLod (sampler2D sampler, vec3 coord, float lod) vec4 texture2DProjLod (sampler2D sampler, vec4 coord, float lod)	
vec4 textureCube (samplerCube sampler, vec3 coord) vec4 textureCube (samplerCube sampler, vec3 coord, float bias) vec4 textureCubeLod (samplerCube sampler, vec3 coord, float lod)	此函数功能为使用纹理坐标coord在由sampler参数指定的立方图纹理中执行纹理采样。此函数中的coord不再是直接的纹理坐标，而是一个表示方向的向量，通过此向量首先确定要采样的面，再确定面里面的纹理坐标
vec4 texture3D	此函数功能是使用纹理坐标coord在由sampler指

(sampler3D sampler, vec3 coord, [float bias]) vec4 texture3DProj (sampler3D sampler, vec4 coord, [float bias]) vec4 texture3DLod (sampler3D sampler, vec3 coord, float lod) vec4 texture3DProjLod (sampler3D sampler, vec4 coord, float lod)	定的3D纹理中执行纹理采样。对于带有Proj后缀的函数，会将纹理坐标（coord.s，coord.t）除以coord的最后一个部分（coord.q）。对于类型为vec4的coord，忽略其第三部分

关于表4-12中的一些内容还需要进行更为详细的一些讨论，具体如下所列。

1．bias参数

含有bias参数的纹理采样函数只能在片元着色器中调用，且此参数仅对sampler为mipmap类型的纹理时才有意义。

- 若提供了bias参数，且sampler对应的纹理为mipmap类型的，则bias参数会用来参与计算细节级别，产生细节级别后再到对应细节级别的mipmap纹理中执行采样。
- 若没有提供bias参数，且sampler对应的纹理为mipmap类型时，将由系统自动计算细节级别，产生细节级别后再到对应细节级别的mipmap纹理中执行采样。

• 若sampler对应的纹理不是mipmap类型时，直接采用原始纹理进行采样。

2．Lod后缀系列

带有“Lod”后缀的纹理采样函数仅适用于顶点着色器，其中的lod（level of detail的缩写）参数将直接用来作为进行mipmap纹理采样时的细节级别。因此，此系列函数也仅对sampler为mipmap类型的纹理才有意义。

3．texture3D系列

texture3D系列的纹理采样函数在标准的OpenGL ES 2.0中还没有支持，但部分厂商的产品进 行了扩展，支持此系列函数。因此，在使用此系列的函数时需要在着色器中打开相应扩展，具体方法为将下面一行代码添加到着色器代码的最开始。

```
#extension GL_OES_texture_3d:enable        //打开扩展后还需要硬件支持才可以真正使用
```

> **提示**
>
> 本小节仅仅是介绍了纹理采样函数的基本知识，纹理采样的相关知识很庞杂，在后继章节中将陆续进行详细的介绍。

4.4.8 微分函数

微分函数仅能用于片元着色器，并且目前还没有纳入OpenGL ES 2.0的正式标准，需要支持特定扩展的硬件才能使用。因此，在使用此系列的函数时需要在着色器中打开相应的扩展，具体方法为将下面一行代码添加到着色器代码的最开始。

```
#extension GL_OES_standard_derivatives:enable
```

OpenGL ES着色语言中的微分函数如表4-13所列。

表4-13　微分函数

内置函数签名	说　明
genType dFdx (genType p)	此函数功能为返回参数p在x方向的偏导数
genType dFdy (genType p)	此函数功能为返回参数p在y方向的偏导数
genType fwidth (genType p)	此函数功能为返回参数p在x与y方向偏导数的绝对值之和，即返回值为abs（dFdx（p））+ abs（dFdy（p））

说明

表4-13中genType代表的数据类型有float、vec2、vec3以及vec4，有关数据类型的详细内容，请读者参考本章4.2.1小节。dFdx与dFdy函数常用来估算过滤器的宽度以及实现抗锯齿等。

一些具体实现中，可以在宿主语言中设置求导的精度，这样开发人员就可以根据需要选择偏重高渲染质量还是高渲染速度，具体的设置代码如下。

```
GLES20.glHint(GL_FRAGMENT_SHADER_DERIVATIVE_HINT, XXXXX);
```

说明

上述代码中的“XXXXX”代表某选项参数的值，目前在Android 2.3及以下版本中还不支持此功能，未来更高的版本应该会支持。

4.5　本章小结

本章简要介绍了与开发酷炫3D场景密切相关的着色语言，主要介绍了着色语言的基础知识、顶点与片元着色器中的内建变量、内置函数等。通过本章的学习，读者应该对着色语言有了一定的了解，能初步使用该语言，为以后开发复杂的、真实的3D场景打下坚实的基础。

第5章　投影及各种变换

3D应用程序开发中，一项很重要的工作就是对场景中的物体进行各种投影与变换。相比于OpenGL ES 1.x的封闭模式，OpenGL ES 2.0在变换方面采取了开放模式，其API中不再提供完成各种变换的方法，变换所用的矩阵都由开发人员直接提供给渲染管线。

因此，基于OpenGL ES 2.0进行投影与变换的开发时，可能需要了解更多的数学知识。这样虽然增加了开发的难度，但大大提高了开发的灵活性，本章将对投影与变换相关的数学知识及具体的实现方法进行详细介绍。

5.1　摄像机的设置

从日常生活的经验中可以很容易地了解到，随着摄像机位置、姿态的不同，就算是对同一个场景进行拍摄，得到的画面也是迥然不同的。因此

摄像机的位置、姿态在OpenGL ES 2.0应用程序的开发中就显得非常重要，故在介绍两种投影与变换之前，首先需要介绍一下摄像机的设置方法。

摄像机的设置主要需要给出3方面的信息，包括摄像机的位置、观察的方向以及up方向，具体情况如图5-1所示。

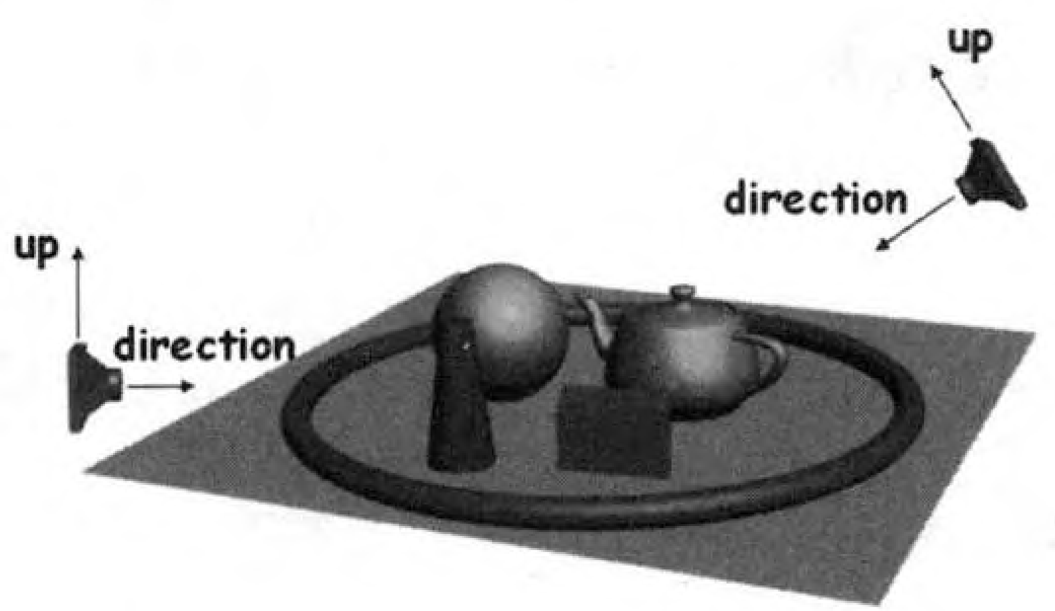


图5-1　摄像机观察物体

- 摄像机的位置很容易理解，用其在3D空间中的坐标来表示。
- 摄像机观察的方向可以理解为摄像机镜头的指向，用一个观察目标点来表示（通过摄像机位置与观察目标点可以确定一个向量，此向量即代表了摄像机观察的方向）。
- 摄像机的up方向可以理解为摄像机顶端的指向，用一个向量来表示。

通过摄像机拍摄场景与人眼观察现实世界很类似，因此通过人眼对现实世界观察的切身感受可以帮助读者理解摄像机的各个参数，如图5-2所示。

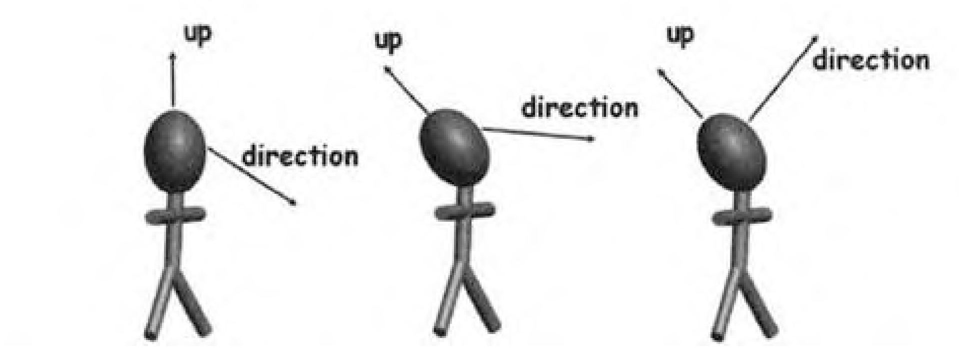


图5-2　人眼观察物体

从图5-2中可以看出，摄像机的位置、朝向、up方向可以有很多不同的组合。例如，同样的位置可以有不同的朝向、不同的up方向；不同的位置也可以具有相同的朝向、相同的up方向等。

本书案例中通过调用Matrix类的setLookAtM方法来完成对摄像机的设置，基本代码如下。

```
Matrix.setLookAtM(
    mVMatrix,              //存储生成矩阵元素的 float[]类型数组
    0,                     //填充起始偏移量
    cx, cy, cz,            //摄像机位置的 X、Y、Z 坐标
    tx, ty, tz,            //观察目标点 X、Y、Z 坐标
    upx, upy, upz          //up 向量在 X、Y、Z 轴上的分量
);
```

说明

从上述代码中可以看出，setLookAtM方法的功能为根据接收的9个摄像机相关参数产生摄像机的观察矩阵，并将矩阵的元素填充到指定的数组中。之所以产生矩阵是因为OpenGL ES 2.0渲染管线的需要，这点在前面已经提到。

5.2　两种投影方式

通过前面章节的学习，读者应该已经了解到，在图元装配之后的光栅化阶段前，首先需要把虚拟3D世界中的物体投影到二维平面上。OpenGL ES 2.0中常用的投影模式有两种，分别为正交投影与透视投影，本节将对这两种投影方式进行详细介绍。

5.2.1 正交投影

OpenGL ES 2.0中，根据应用程序中提供的投影矩阵，管线会确定一个可视空间区域，称之为视景体。视景体是由6个平面确定的，这6个平面分别为：上平面（up）、下平面（down）、左平面（left）、右平面（right）、远平面（far）、近平面（near）。

场景中处于视景体内的物体会被投影到近平面上（视景体外面的物体将被裁剪掉），然后再将近平面上投影出的内容映射到屏幕上的视口中。对于正交投影而言，视景体及近平面的情况如图5-3所示。

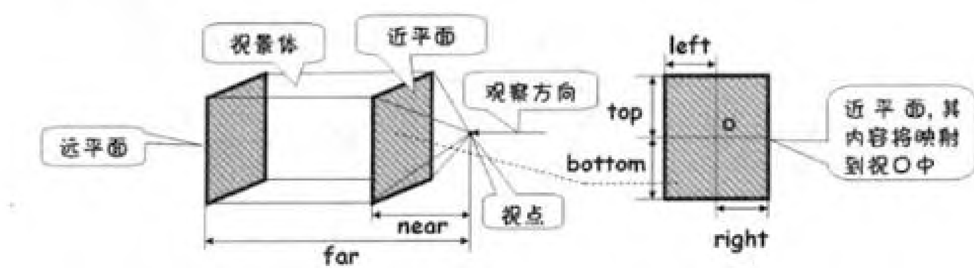


图5-3　正交投影示意图

> **说明**
>
> 视点为摄像机的位置；离视点较近（距离为near），垂直于观察方向向量的平面为近平面，离视点较远（距离为far），垂直于观察方向向量的平面为远平面。与观察向量平行，从上下左右4个方向约束视景体范围的4个平面分别为上平面、下平面、左平面、右平面，这4个平面与视景体中心轴线的距离分别为top、bottom、left、right。

从图5-3中可以看出，由于正交投影是平行投影的一种，其投影线（物体的顶点与近平面上投影点的连线）是平行的。故其视景体为长方体，投影到近平面上的图形不会产生真实世界中“近大远小”的效果，图5-4更清楚地说明了这个问题。

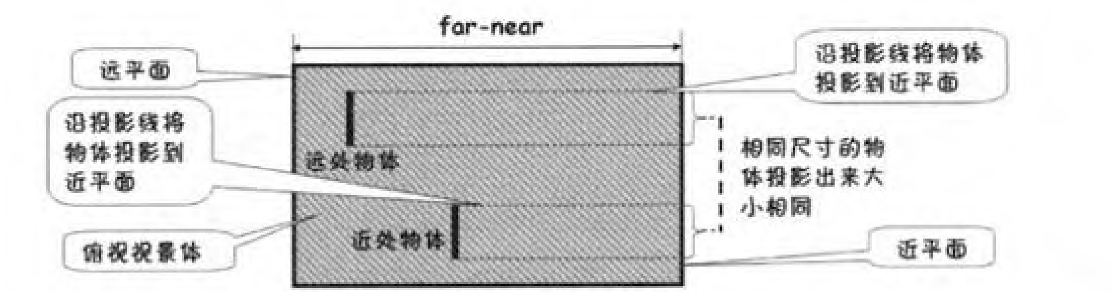


图5-4　正交投影不产生"近大远小"效果的原理图

本书案例中通过调用Matrix类的orthoM方法完成对正交投影的设置，基本代码如下。

```
Matrix.orthoM(
        mProjMatrix,                //存储生成矩阵元素的float[]类型数组
        0,                          //填充起始偏移量
        left, right,                //near面的left、right
        bottom, top,                //near面的bottom、top
        near, far                   //near面、far面与视点的距离
);
```

• orthoM方法的功能为根据接收的6个正交投影相关参数产生正交投影矩阵，并将矩阵的元素填充到指定的数组中。

• 参数left、right为近平面左右侧边对应的x坐标，bottom、top为近平面上下侧边对应的y坐标，分别用来确定左平面、右平面、下平面、上平面的位置。参数near、far分别为视景体近平面与远平面距视点的距离。

前面提到过，场景中的物体投影到近平面后，最终会映射到显示屏上的视口中。视口也就是显示屏上指定的矩形区域，通过如下代码进行设置。

```
GLES20.glViewport(x, y, width, height);            //设置视口
```

上述代码中参数x、y为视口矩形左上侧点在屏幕坐标系内的坐标，width、heighi为视口的宽度与高度，具体情况如图5-5所示。

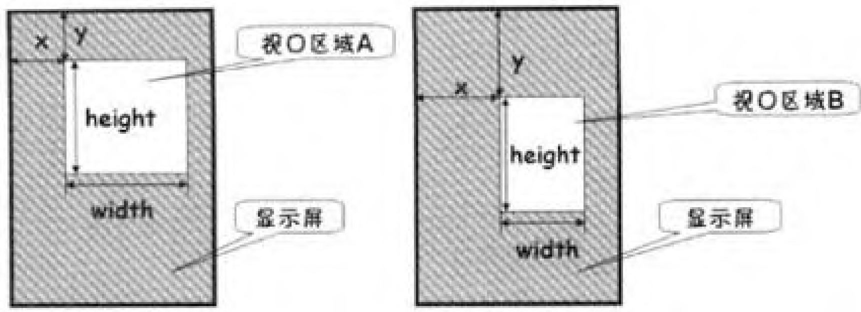


图5-5　视口的示意图

> **提示**
>
> 从近平面到视口的映射是由渲染管线自动完成的。一般情况下，应该保证近平面的宽高比与视口的宽高比相同，也就是满足“（left+right）/（top+bottom）==width/height”，否则显示在屏幕上的图像会拉伸变形。

前面介绍了正交投影的基本知识，下面将给出一个使用了正交投影的小案例——Sample5_1，其运行效果如图5-6所示。

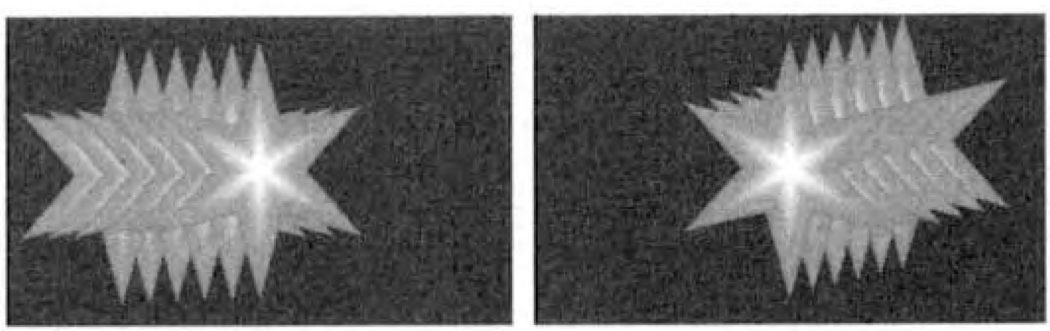


图5-6　案例Sample5_1的运行效果图

> **提示**
>
> 从图5-6中可以看出，本案例的场景是由一组距离观察点越来越远的相同尺寸的六角星构成。由于采用的投影方式为正交投影，因此最终显示在屏幕上的每个六角星尺寸都相同。

了解了本案例的运行效果后，下面就可以进行代码的开发了，具体步骤如下。

（1）首先需要开发案例的主控制类——Sample5_1_Activity。由于本案例中的Activity类与3.1.2节Sample3_1中的Activity类非常相似，这里就不再赘述其代码。需要的读者请参考3.1.2节中对Activity类的讲解，自行查看随书光盘中的源代码。同时，本案例中用到的顶点与片元着色器代码也与3.1.2节案例中的相同，这里也不再赘述。

（2）完成了Sample5_1_Activity类的开发后，接下来将开发用来渲染3D场景的MySurfaceView类，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_1/com/bn/Sample5_1目录下的MySurfaceView.java。

```
package com.bn.Sample5_1;                                   //声明包
import android.opengl.GLSurfaceView;                        //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.content.Context;                             //引入相关类
class MySurfaceView extends GLSurfaceView {
```

```
6   private final float TOUCH_SCALE_FACTOR = 180.0f / 320;   // 角度缩放比例
7   private SceneRenderer mRenderer;                         // 场景渲染器
8   private float mPreviousY;                                // 上次的触控位置 Y 坐标
9   private float mPreviousX;                                // 上次的触控位置 X 坐标
10      public MySurfaceView(Context context) {
11        ……/*此处省略了与本书前面案例相似类中完全相同的部分代码,
12          需要的读者可以自行查看随书光盘中的源代码*/ }
13      @Override
14      public boolean onTouchEvent(MotionEvent e) {          //触摸事件回调方法
15        float y = e.getY();  float x = e.getX();            //获取此次触控的 X、Y 坐标
16        switch (e.getAction()) {
17        case MotionEvent.ACTION_MOVE:                       //若为移动动作
18          float dy = y - mPreviousY;float dx = x - mPreviousX;//计算触控位置的 X、Y 位移
19          for(SixPointedStar h:mRenderer.ha) {  //设置各个六角星绕 X 轴、Y 轴旋转的角度
20            h.yAngle += dx * TOUCH_SCALE_FACTOR;
21            h.xAngle+= dy * TOUCH_SCALE_FACTOR;
22        } }
23        mPreviousY = y; mPreviousX = x; //记录触控笔的 X、Y 坐标
24        return true;
25      }
26      private class SceneRenderer implements GLSurfaceView.Renderer {
27          SixPointedStar[] ha = new SixPointedStar[6];      //六角星数组
28          public void onDrawFrame(GL10 gl) {
29              GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT     //清除深度缓冲与颜色缓冲
30                      | GLES20.GL_COLOR_BUFFER_BIT);
31              for (SixPointedStar h : ha) {                 //循环绘制各个六角星
32                  h.drawSelf();
33          }}
34          public void onSurfaceChanged(GL10 gl, int width, int height) {
35              GLES20.glViewport(0, 0, width, height);       //设置视口的大小及位置
36              float ratio = (float) width / height;         //计算视口的宽高比
37              MatrixState.setProjectOrtho(-ratio, ratio, -1, 1, 1, 10);//设置正交投影
38              MatrixState.setCamera(0, 0, 3f, 0, 0, 0f, 0f, 1.0f, 0.0f); //设置摄像机
39          }
40          public void onSurfaceCreated(GL10 gl, EGLConfig config) {
41              GLES20.glClearColor(0.5f, 0.5f, 0.5f, 1.0f);//设置屏幕背景色 RGBA
42              for (int i = 0; i < ha.length; i++) {   //创建六角星数组中的各个六角星
43                  ha[i] = new SixPointedStar(MySurfaceView.this, 0.2f, 0.5f, -0.3f * i);
44              }
45              GLES20.glEnable(GLES20.GL_DEPTH_TEST);        // 打开深度检测
46  }}}
```

- 第14-25行为用于处理屏幕触控事件的onTouchEvent方法，其功能为根据触控点的X、Y位移改变各个六角星绕X轴及Y轴的旋转角度。

- 第26-46行为场景渲染器SceneRenderer的代码，其在onSurfaceCreated方法中创建了六角星数组中的各个六角星对象。从代码中可以看出，创建时依次为六角星指定了离观察点越来越远的Z坐标。
- 第34-39行在onSurfaceChanged方法中调用了GLES20类的glViewport方法设置视口，调 用了MatrixState类中的setProjectOrtho方法设置了正交投影。从上述两个方法的参数中可以看出，视口的长宽比与近平面的长宽比是相同的，这样显示在屏幕上的内容才不会变形。
- 第28-33行为onDrawFrame方法的实现，其在清除深度缓冲与颜色缓冲之后，遍历六角星数组，通过调用每个六角星对象的drawSelf方法来绘制各个六角星。

提示

上述代码中用到的MatrixState类是笔者自己开发的用于封装Matrix类各项功能以简化开发的工具类，后面将进行详细介绍。

（3）前面的步骤中使用了SixPointedStar类来创建六角星对象，下面将介绍此类的开发，其具体代码框架如下。

代码位置：见随书光盘中源代码/第5章/Sample5_1/com/bn/Sample5_1目录下的SixPointedStar.java。

```
1 package com.bn.Sample5_1;                                        //声明包
2 import static com.bn.Sample5_1.ShaderUtil.createProgram;         //引入相关类
3 ……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
4 import android.opengl.Matrix;                                    //引入相关类
5 public class SixPointedStar {                                    //六角星
6     …… //省略一些成员变量的声明，需要的读者参考光盘中的源代码
7     float yAngle = 0; float xAngle = 0;                          //绕 X 轴、Z 轴旋转的角度
8     final float UNIT_SIZE = 1;
9     public SixPointedStar(MySurfaceView mv, float r, float R, float z) {
10            initVertexData(R, r, z);        //调用初始化顶点数据的 initVertexData 方法
11            initShader(mv);                 //调用初始化着色器的 initShader 方法
12        }
13        public void initVertexData(float R, float r, float z) {  //初始化顶点数据的
      initVertexData 方法
14            ……//此处省略了初始化顶点数据的方法，将在下面介绍
15        }
16        public void initShader(MySurfaceView mv) {        //初始化着色器的 initShader 方法
17            ……//此处省略了部分代码，与前面案例的基本相同，读者可自行查看随书光盘中的源代码
18        }
19        public void drawSelf() {                          //绘制图形的方法
20            ……//此处省略了绘制图形的方法，将在下面介绍
21    }}
```

- 第6-12行为该类中成员变量和构造器的代码，各成员变量与前面案例Sample3_1里的Triangle类非常类似。只是减少了投影矩阵、摄像机位置朝向参数矩阵和总变换矩阵的声明，这些矩阵已经封装在了后面将要开发的MatrixState类中。
- 第13-15行为初始化顶点数据的initVertexData方法，将在下面介绍。
- 第19-21行为绘制六角星的drawSelf方法，也将在下面介绍。

提示

第16-18行的initShader方法与Sample3_1中Triangle类里的initShader方法基本一致，这里就不再赘述。后面本书的案例中，将只讲解与前面案例的不同之处。前面案例中已经出现过的代码，将进行适当的省略以节省篇幅，需要的读者可以查看光盘中的源代码。

（4）完成了SixPointedStar类代码框架的开发后，下面将开发上一步骤中省略的初始化顶点数据的initVertexData方法，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_1/com/bn/Sample5_1目录下的SixPointedStar.java。

```
public void initVertexData(float R, float r, float z) { //初始化顶点数据的 initVertexData 方法
    List<Float> flist = new ArrayList<Float>();
    float tempAngle = 360 / 6;
    for (float angle = 0;angle <360; angle+= tempAngle) {
                                               //循环生成构成六角形各三角形的顶点坐标
        flist.add(0f); flist.add(0f); flist.add(z);      //第一个点的 x、y、z 坐标
        flist.add((float) (R * UNIT_SIZE * Math.cos(Math.toRadians(angle))));
                                               //第二个点的 x 坐标
        flist.add((float) (R * UNIT_SIZE * Math.sin(Math.toRadians(angle))));
                                               //第二个点的 y 坐标
        flist.add(z);                          //第二个点的 z 坐标
        flist.add((float) (r * UNIT_SIZE * Math.cos(Math.toRadians(angle
                                               //第三个点的 x 坐标
                + tempAngle / 2))));
            flist.add((float) (r * UNIT_SIZE * Math.sin(Math.toRadians(angle
                                               //第三个点的 y 坐标
                + tempAngle / 2))));
            flist.add(z);                      //第三个点的 z 坐标
            flist.add(0f);flist.add(0f);flist.add(z);      //第一个中心点的 x、y、z 坐标
            flist.add((float) (r * UNIT_SIZE * Math.cos(Math.toRadians(angle
                                               //第二个点的 x 坐标
                + tempAngle / 2))));
            flist.add((float) (r * UNIT_SIZE * Math.sin(Math.toRadians(angle
                                               //第二个点的 y 坐标
                + tempAngle / 2))));
            flist.add(z);                      //第二个点的 z 坐标
            flist.add((float) (R * UNIT_SIZE * Math.cos(Math.toRadians(angle
                                               //第三个点的 x 坐标
                + tempAngle))));
            flist.add((float) (R * UNIT_SIZE * Math.sin(Math.toRadians(angle
                                               //第三个点的 y 坐标
                + tempAngle))));
            flist.add(z);                      //第三个点的 z 坐标
        }
        vCount = flist.size() / 3;
        float[] vertexArray = new float[flist.size()];      //顶点坐标数组
        for (int i = 0; i < vCount; i++) {
            vertexArray[i * 3] = flist.get(i * 3);
            vertexArray[i * 3 + 1] = flist.get(i * 3 + 1);
            vertexArray[i * 3 + 2] = flist.get(i * 3 + 2);
        }
        ByteBuffer vbb = ByteBuffer.allocateDirect(vertexArray.length * 4);
        vbb.order(ByteOrder.nativeOrder());    //设置字节顺序为本地操作系统顺序
        mVertexBuffer = vbb.asFloatBuffer();
        mVertexBuffer.put(vertexArray);        //将顶点坐标数据放进缓冲
        mVertexBuffer.position(0);
```

```
38        float[] colorArray = new float[vCount * 4];     //顶点着色数据的初始化
39        for (int i = 0; i < vCount; i++) {
40            if (i % 3 == 0) {              //中心点为白色，RGBA 4个通道[1,1,1,0]
41                colorArray[i * 4] = 1;colorArray[i * 4 + 1] = 1;
42                colorArray[i * 4 + 2] = 1;colorArray[i * 4 + 3] = 0;
43            } else {              //边上的点为淡蓝色,RGBA 4个通道[0.45,0.75,0.75,0]
44                colorArray[i * 4] = 0.45f;colorArray[i * 4 + 1] = 0.75f;
45                colorArray[i * 4 + 2] = 0.75f;colorArray[i * 4 + 3] = 0;
46            }
47        }
48        ByteBuffer cbb = ByteBuffer.allocateDirect(colorArray.length * 4);
49        cbb.order(ByteOrder.nativeOrder());             //设置字节顺序为本地操作系统顺序
50        mColorBuffer = cbb.asFloatBuffer();
51        mColorBuffer.put(colorArray);                   //将顶点颜色数据放进缓冲
52        mColorBuffer.position(0);
53    }
```

- 第2-37行初始化了六角星的顶点位置数据。基本思想为将一个圆周平均分为6段，每段60°，包含一个四边形。然后依次将构成每个四边形的两个三角形的3个顶点坐标计算出来，放进顶点坐标数组，最后再将所有的坐标数据放进缓冲。
- 第38-52行初始化了六角星的顶点颜色数据，其中将位于中心的顶点设置为白色，其余顶点设置为淡蓝色，然后将所有的颜色数据放入缓冲区。需要注意的是每个顶点的颜色需要4个数值，分别代表R（红色）、G（绿色）、B（蓝色）、A（透明度）4个色彩通道，例如[1，1，1，0]就是白色。

提示

从上述代码中可以看出构成一个六角星共使用了12个三角形（6个四边形，每个四边形两个三角形），合计12×3（36）个顶点。因此一共有36套顶点坐标数据，每套3个坐标值（*X*、*Y*、*Z* ）。同时就需要36套颜色数据，每套4个色彩通道值（R、G、B、A）。读者在开发时也需要注意顶点的各项数据数量要匹配，并一一对应，否则可能造成程序不能正确渲染场景。

（5）完成了initVertexData方法的开发后，就可以开发用来渲染六角星的drawSelf方法了，其具体代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_1/com/bn/Sample5_1目录下的SixPointedStar.java。

```java
public void drawSelf() {
    GLES20.glUseProgram(mProgram);                          //指定使用某套着色器程序
    Matrix.setRotateM(mMMatrix, 0, 0, 0, 1, 0);             //初始化变换矩阵
    Matrix.translateM(mMMatrix, 0, 0, 0, 1);                //设置沿Z轴正向位移1
    Matrix.rotateM(mMMatrix, 0, yAngle, 0, 1, 0);           //设置绕y轴旋转yAngle度
    Matrix.rotateM(mMMatrix, 0, xAngle, 1, 0, 0);           //设置绕x轴旋转xAngle度
    GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1,         //将最终变换矩阵传入渲染管线
            false,MatrixState.getFinalMatrix(mMMatrix), 0);
    GLES20.glVertexAttribPointer(maPositionHandle, 3,       //将顶点位置数据传送进渲染管线
                GLES20.GL_FLOAT, false, 3 * 4, mVertexBuffer);
        GLES20.glVertexAttribPointer(maColorHandle, 4,      //将顶点颜色数据传送进渲染管线
                GLES20.GL_FLOAT, false, 4 * 4, mColorBuffer);
        GLES20.glEnableVertexAttribArray(maPositionHandle);      //启用顶点位置数据
        GLES20.glEnableVertexAttribArray(maColorHandle);         //启用顶点着色数据
        GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount);     //绘制六角星
}
```

说明

此方法的功能为首先进行恰当的变换，然后将最终变换矩阵、顶点位置数据、顶点颜色数据送入渲染管线，最后渲染六角星。其中用到了笔者自己开发的MatrixState类，将在后面介绍。

（6）前面不少步骤中都用到了MatrixState类中的工具方法，下面将进行此类的开发，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_1/com/bn/Sample5_1目录下的MatrixState.java。

```
1 package com.bn.Sample5_2;                              //声明包
2 import android.opengl.Matrix;                          //引入相关类
3 public class MatrixState {                             //存储系统矩阵状态的类
4     private static float[] mProjMatrix = new float[16]; //4x4矩阵 投影用
5     private static float[] mVMatrix = new float[16];    //摄像机位置朝向9参数矩阵
6     private static float[] mMVPMatrix;                  //最终的总变换矩阵
7     public static void setCamera(                       //设置摄像机的方法
8             float cx, float cy, float cz,               //摄像机位置的X、Y、Z坐标
9             float tx, float ty, float tz,               //观察目标点X、Y、Z坐标
10                float upx, float upy, float upz         //up向量在X、Y、Z轴上的分量
11        ) {
12            Matrix.setLookAtM(mVMatrix, 0, cx, cy, cz, tx, ty, tz, upx, upy, upz);
13        }
14        public static void setProjectOrtho(             //设置正交投影的方法
15                float left, float right,                //near面的left、right
16                float bottom, float top,                //near面的bottom、top
17                float near, float far                   //near面、far面与视点的距离
18        ) {
19            Matrix.orthoM(mProjMatrix, 0, left, right, bottom, top, near, far);
20        }
21        public static float[] getFinalMatrix(float[] spec) { //生成物体总变换矩阵的方法
22            mMVPMatrix = new float[16];                 //创建用来存放最终变换矩阵的数组
23            Matrix.multiplyMM(mMVPMatrix, 0, mVMatrix, 0, spec, 0);
                                                          //将摄像机矩阵乘以变换矩阵
24            //将投影矩阵乘以上一步的结果矩阵得到最终变换矩阵
25            Matrix.multiplyMM(mMVPMatrix, 0, mProjMatrix, 0, mMVPMatrix, 0);
26            return mMVMatrix;
27    }}
```

- 第4-6行声明了用来存放投影矩阵、摄像机矩阵、最终变换矩阵的3个数组，供后面用来生成相应矩阵的设置方法使用。

- 第7-13行为设置摄像机的setCamera方法，其功能为根据接收的摄像机9个相关参数生成摄像机观察矩阵，并将矩阵元素依次存放到mVMatrix成员数组中。
- 第14-20行为设置正交投影的setProjectOrtho方法，其功能为根据接收的正交投影6个相关参数生成正交投影矩阵，并将矩阵元素依次存放到mProjMatrix成员数组中。
- 设置摄像机的setCamera方法以及设置正交投影的setProjectOrtho方法都是通过调用Matrix类的相关方法完成的，因此这两个方法主要是提供了对Matrix类中相关方法的封装。
- 第21-27行为根据变换矩阵、摄像机观察矩阵、投影矩阵生成最终变换矩阵的方法。此方法通过依次将投影矩阵、摄像机观察矩阵、变换矩阵相乘来产生最终变换矩阵。“变换矩阵”是指用来完成3D场景中物体的平移、旋转、缩放等变换的矩阵，其相关知识将在本章后继的内容中详细讨论。

提示

MatrixState类是本书中非常重要的一个类，几乎本书中所有的案例都要使用到，本章后继的章节将不断对此类进行完善。之所以要开发此类是因为，若将对矩阵的很多操作散落在各类的代码中有很多缺点，如不易管理、不易维护、代码不能有效重用等，这些读者在后面的进一步学习中会逐渐感觉到。

5.2.2　透视投影

现实世界中人眼观察物体时会有“近大远小”的效果，因此要想开发出更加真实的场景，仅使用正交投影是远远不够的，这时可以采用透视投影。透视投影的投影线是不平行的，他们相交于视点。通过透视投影，可以产生现实世界中“近大远小”的效果，大部分3D游戏采用的都是透视投影。

透视投影中，视景体为锥台形区域，如图5-7所示。

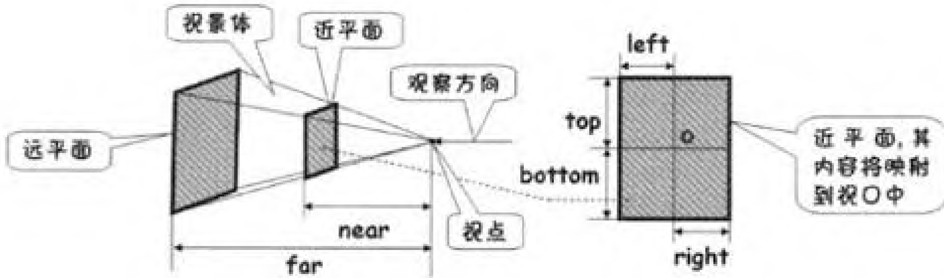


图5-7　透视投影示意图

> **说明**
>
> 视点为摄像机的位置；离视点较近（距离为near），垂直于观察方向向量的平面为近平面；离视点较远（距离为far），垂直于观察方向向量的平面为远平面。近平面左侧距中心的距离为left，右侧为right，上侧为top，下侧为bottom。由观察点与近平面左上、左下、右上、右下4个的点连线与远平面的交点可以确定上、下、左、右4个斜面，这4个斜面及远近平面确定了视景体的范围。

从图5-7中可以看出，透视投影的投影线互不平行，都相交于视点。因此同样尺寸的物体，近处的投影出来大，远处的投影出来小，从而产生了现实世界中“近大远小”的效果，图5-8更清楚地说明了这个问题。

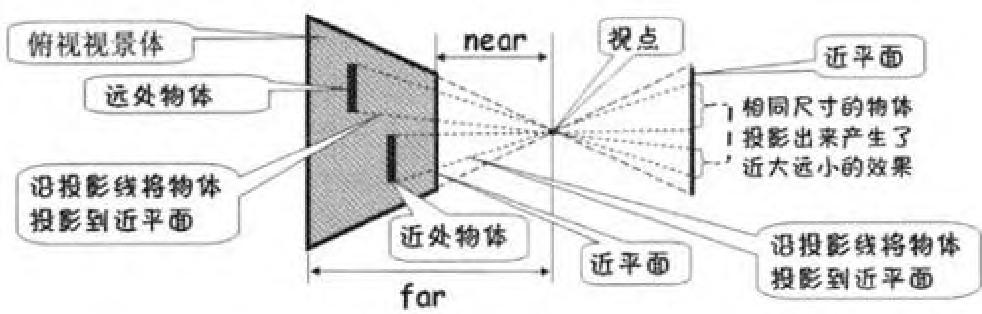


图5-8　透视投影产生“近大远小”效果的原理图

本书案例中通过调用Matrix类的frustumM方法完成对透视投影的设置，基本代码如下。

```
Matrix.frustumM(
        mProjMatrix,                //要填充矩阵元素的float[]类型数组
        0,                          //填充起始偏移量
        left, right,                //near面的left、right
        bottom, top,                //near面的bottom、top
        near, far                   //near面、far面与视点的距离
);
```

说明

从上述代码中可以看出，frustumM方法的功能为根据接收的6个透视投影相关参数产生透视投影矩阵，并将矩阵的元素依次填充到指定的数组中。

前面介绍了透视投影的基本知识，下面将给出一个使用了透视投影的小案例——Sample5_2，其运行效果如图5-9所示。

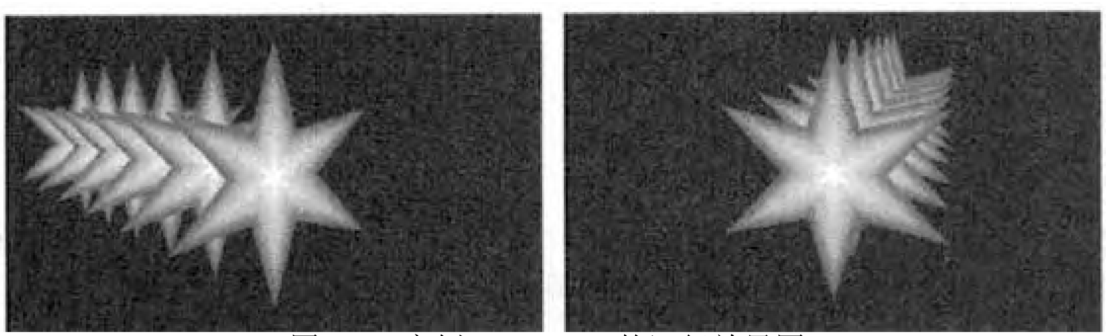


图5-9　案例Sample5_2的运行效果图

提示

与Sample5_1相同，本案例的场景是由一组距离观察点越来越远的相同尺寸的六角星构成。由于采用的投影方式为透视投影，因此最终显示在屏幕上的多个六角星近大远小。

了解了本案例的运行效果后，下面就可以进行代码的开发了，具体步骤如下。

提示

由于本案例中大部分代码与Sample5_1完全相同，因此下面的步骤仅介绍代码有区别的部分。

（1）在MatrixState类中增加了设置透视投影的setProjectFrustum方法，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_2/com/bn/Sample5_2目录下的MatrixState.java。

```
public static void setProjectFrustum(                   //设置透视投影
        float left, float right,                        //near面的left和right
        float bottom, float top,                        //near面的bottom和top
        float near, float far                           // near面、far面与视点的距离
) {
    Matrix.frustumM(mProjMatrix, 0, left, right, bottom, top, near, far);
}
```

提示

设置透视投影的方法与设置正交投影的方法类似，都是对Matrix类中相关方法的封装。

（2）在MySurfaceView类的onSurfaceChanged方法中，将设置正交投影的语句替换为设置透视投影的语句，具体代码如下。

代码位置：见随书光盘中源代码，第5章/Sample5_2/com/bn/Sample5_2目录下的MySurfaceView.java。

```java
1 public void onSurfaceChanged(GL10 gl, int width, int height) {
2     GLES20.glViewport(0, 0, width, height);          //设置视口的大小及位置
3     float ratio= (float) width / height;             //计算视口的宽高比
4     MatrixState.setProjectFrustum(-ratio*0.4f, ratio*0.4f, -1*0.4f, 1*0.4f, 1, 50);
                                                       //设置透视投影
5     MatrixState.setCamera(0,0,6,0f,0f,0f,0f,1.0f,0.0f); //设置摄像机
6 }
```

提示

第4行将Sample5_1中设置正交投影的语句替换成了设置透视投影的语句，其他的基本没有变化。要注意的是，与前面的案例相同也要保持近平面与视口的宽高比一致，否则最终显示到屏幕上的物体会拉伸变形。

（3）在MySurfaceView类的onSurfaceCreated方法中，创建六角星数组里的各个六角星对象时，适当调整了各个六角星的位置，具体代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_2/com/bn/Sample5_2目录下的MySurfaceView.java。

```java
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    GLES20.glClearColor(0.5f, 0.5f, 0.5f, 1.0f);      // 设置屏幕背景色RGBA
    for (int i = 0; i < ha.length; i++) {             // 创建六角星数组中的各个对象
        ha[i] = new SixPointedStar(MySurfaceView.this, 0.4f, 1.0f,-1.0f * i);
    }
    GLES20.glEnable(GLES20.GL_DEPTH_TEST);            // 打开深度检测
}
```

提示

第4行在创建各个六角星对象时，将最后一个参数由Sample5_1中的-0.3f*i换成了-1.0f*i，使每个六角星离视点更远，这使得近大远小的效果更加明显。另外实际开发中摆放物体时，要注意只有位于视景体内部的物体才能看见，将物体摆放得太近（位于近平面与视点之间）而看不见是初学者常犯的错误之一，读者要多加注意。

5.3　各种变换

上一小节六角星的案例中已经用到了物体的平移及旋转，但没有进行详细的介绍。本节将对3种基本变换（包括平移、旋转、缩放）的相关理论知识及具体开发进行介绍。同时也会进一步完善贯穿本书、几乎所有案例都需要使用的MatrixState类，为其增加实现各种变换的方法。

5.3.1　基本变换的相关数学知识

基本变换都是通过将表示点坐标的向量与特定的变换矩阵相乘完成的，进行基于矩阵的变换时，三维空间中点的位置需要表示成齐次坐标形式。所谓齐次坐标形式也就是在X、Y、Z 3个坐标值后面增加第四个量w，未变换时w值一般为1，如P=（P_x，P_y，P_z，1）^{T}。

P与一个特定的变换矩阵M相乘即可以完成一次基本变换，得到变换后点Q的齐次坐标向量，如Q（Q_x，Q_y，Q_z，1）^{T}，具体情况如下所列。

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

或简写为：Q=MP

说明

上述线性代数表达式中最左侧为变换后Q点的齐次坐标，中间的为4×4的变换矩阵，右侧为变换前P点的齐次坐标。

当矩阵M中的元素取适当的值时，等式Q=MP就会有其特殊的几何意义。例如，可以将三维空间中的点P平移、旋转或缩放到点Q。这些变换的具体

信息就存放在矩阵M中，因此通常称矩阵M为变换矩阵。当需要连续执行一系列的变换时，依次将变换矩阵乘以表示点位置的齐次坐标向量即可。

> **提示**
>
> 数学上，向量表示可以有两种选择：行向量与列向量。这两种方式没有本质区别，选取哪种都可以，OpenGL ES中使用的是列向量。列向量和矩阵相乘实现变换时，只能在列向量前面乘以矩阵，而行向量则反之，否则乘法没有意义，读者开发时要注意这一点。由于本书篇幅有限，在此仅简单介绍了基本变换的数学原理。有关矩阵和向量的更多知识，请读者参考其他线性代数的相关书籍或资料。

5.3.2 平移变换

前面的小节已经介绍过，OpenGL ES中的基本变换都是通过变换矩阵完成的，平移变换自然也是如此，其变换矩阵的基本格式如下。

$$M=\begin{pmatrix}1 & 0 & 0 & m_x\\ 0 & 1 & 0 & m_y\\ 0 & 0 & 1 & m_z\\ 0 & 0 & 0 & 1\end{pmatrix}$$

上述矩阵中的m_x、m_y、m_z分别表示平移变换中沿X、Y、Z轴方向的位移。通过简单的线性代数计算即可验证，矩阵M乘以变换前P点的齐次坐标后确实得到了相当于将P点沿X、Y、Z轴平移m_x、m_y、m_z的结果，具体情况如下。

$$MP=\begin{pmatrix}1 & 0 & 0 & m_x\\ 0 & 1 & 0 & m_y\\ 0 & 0 & 1 & m_z\\ 0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix}P_x\\ P_y\\ P_z\\ 1\end{pmatrix}=\begin{pmatrix}P_x+m_x\\ P_y+m_y\\ P_z+m_z\\ 1\end{pmatrix}，即\begin{pmatrix}Q_x\\ Q_y\\ Q_z\\ 1\end{pmatrix}=\begin{pmatrix}P_x+m_x\\ P_y+m_y\\ P_z+m_z\\ 1\end{pmatrix}$$

了解了平移变换矩阵的基本情况后，下面给出一个使用了平移变换的简单案例——Sample5_3，其运行效果如图5-10所示。

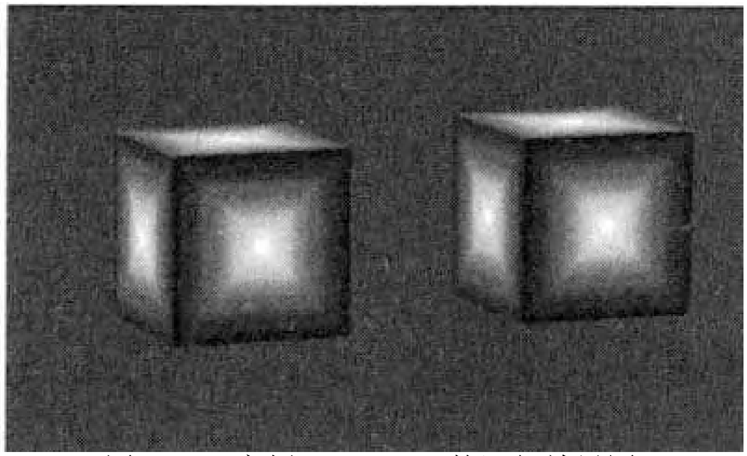


图5-10　案例Sample5_3的运行效果图

了解了本案例的运行效果后，下面就可以进行代码的开发了，具体步骤如下。

（1）首先需要开发本案例的主控制类——Sample5_3_Activity。由于本案例中的Activity类与前面案例中的非常相似，这里就不再赘述，需要的读者请自行查看随书光盘中的源代码。

（2）完成了Sample5_3_Activity类的开发后，接下来需要开发的是用来渲染3D场景的MySurfaceView类，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_3/com/bn/Sample5_3目录下的MySurfaceView.java。

```
package com.bn.Sample5_3;                                    //声明包
import android.opengl.GLSurfaceView;                         //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import static com.bn.Sample5_3.Constant.*;                   //引入相关类
class MySurfaceView extends GLSurfaceView {
    private SceneRenderer mRenderer;                         //场景渲染器
    public MySurfaceView(Context context) {
        ……//此处省略了构造器中的代码，读者可自行查看随书光盘的源代码
    }
        private class SceneRenderer implements GLSurfaceView.Renderer {
            Cube cube;                                       //立方体对象引用
            public void onDrawFrame(GL10 gl) {
                GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT//清除深度缓冲与颜色缓冲
                        | GLES20.GL_COLOR_BUFFER_BIT);
                //绘制原立方体
                MatrixState.pushMatrix();                    //保护现场
```

```
17              cube.drawSelf();                              //绘制立方体
18              MatrixState.popMatrix();                      //恢复现场
19              //绘制变换后的立方体
20              MatrixState.pushMatrix();                     //保护现场
21              MatrixState.translate(4, 0, 0);               //沿x轴方向平移4
22              cube.drawSelf();                              //绘制立方体
23              MatrixState.popMatrix();                      //恢复现场
24          }
25          public void onSurfaceChanged(GL10 gl, int width, int height) {
26              GLES20.glViewport(0, 0, width, height);       //设置视口大小及位置
27              Constant.ratio = (float) width / height;      //计算视口的宽高比
28              MatrixState.setProjectFrustum(            //调用此方法产生透视投影矩阵
29                      -Constant.ratio*0.8f, Constant.ratio*1.2f, -1, 1, 20, 100);
30              MatrixState.setCamera(                        //调用此方法产生摄像机矩阵
31                      -16f, 8f, 45, 0f, 0f, 0f, 0f, 1.0f, 0.0f);
32              MatrixState.setInitStack();                   //初始化变换矩阵
33          }
34          public void onSurfaceCreated(GL10 gl, EGLConfig config) {
35              GLES20.glClearColor(0.5f, 0.5f, 0.5f, 1.0f);//设置屏幕背景色RGBA
36              cube = new Cube(MySurfaceView.this);          //创建立方体对象
37              GLES20.glEnable(GLES20.GL_DEPTH_TEST);        //打开深度检测
38              GLES20.glEnable(GLES20.GL_CULL_FACE);         //打开背面剪裁
39  }}}
```

- 第12-24行为渲染每帧画面的onDrawFrame方法，其中在清除深度缓冲与颜色缓冲之后，绘制了两次立方体。第一次在原位绘制，第二次沿X轴方向进行平移变换后绘制。每次绘制前都调用了MatrixState类的pushMatrix方法保护现场，绘制后都调用MatrixState类的popMatrix方法恢复了现场，具体情况在后面进行介绍。

- 第32行在初始化的onSurfaceChanged方法中调用了MatrixState类的setlnitStack初始化了变换矩阵，具体情况在后面进行介绍。
- 第37-38行分别打开深度检测与背面剪裁，关于深度检测与背面剪裁的详细知识将在后面的章节中陆续介绍。

> **提示**
>
> 本案例中的立方体类——Cube，ShaderUtil类、顶点着色器及片元着色器的代码和前面章节案例中的非常相似，这里不再赘述，请读者自行查看随书光盘中的源代码。

（3）接下来介绍如何在MatrixState类中增加平移变换的功能方法，并升级MatrixState类具有保存与恢复变换矩阵状态的功能。升级后的MatrixState类代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_3/com/bn/Sample5_3目录下的MatrixState.java。

```
package com.bn.Sample5_3;                    //声明包
import java.nio.ByteBuffer;                  //引入相关类
import android.opengl.Matrix;                //引入相关类
```

```
4 public class MatrixState {                              //管理系统变换矩阵的类
5     private static float[] mProjMatrix = new float[16]; //4x4 矩阵 投影用
6     private static float[] mVMatrix = new float[16];    //摄像机位置朝向 9 参数矩阵
7     private static float[] currMatrix;                  //当前变换矩阵
8     static float[][] mStack = new float[10][16];        //用于保存变换矩阵的栈
9     static int  stackTop = -1;                          //栈顶索引
10        public static void setInitStack() {             //产生无任何变换的初始矩阵
11            currMatrix = new float[16];
12            Matrix.setRotateM(currMatrix, 0, 0, 1, 0, 0);
13        }
14        public static void pushMatrix() {               //将当前变换矩阵存入栈中
15            stackTop++;                                 //栈顶索引加 1
16            for (int i = 0; i < 16; i++) {              //当前变换矩阵中的各元素入栈
17                mStack[stackTop][i] = currMatrix[i];
18        }}
19        public static void popMatrix() {                //从栈顶取出变换矩阵
20            for (int i = 0; i < 16; i++) {              //栈顶矩阵元素进当前变换矩阵
21                currMatrix[i] = mStack[stackTop][i];
22            }
23            stackTop--;                                 //栈顶索引减 1
24        }
25        public static void translate(float x, float y, float z) {
                                                      //沿 X、Y、Z 轴方向进行平移变换的方法
26            Matrix.translateM(currMatrix, 0, x, y, z);
27        }
28        static float[] mMVPMatrix = new float[16];          //总变换矩阵
29        public static float[] getFinalMatrix() {            //计算产生总变换矩阵的方法
30            Matrix.multiplyMM(mMVPMatrix, 0, mVMatrix, 0, currMatrix, 0);
                                                              //摄像机矩阵乘以变换矩阵
31            //投影矩阵乘以上一步的结果矩阵
32            Matrix.multiplyMM(mMVPMatrix, 0, mProjMatrix, 0, mMVPMatrix, 0);
33            return mMVPMatrix;
34        }
35        ……//此处省略了部分成员变量和方法，读者可自行查看随书光盘的源代码
36    }
```

- 第5-9行为成员变量的声明，比升级前增加了当前变换矩阵currMatrix、保存变换矩阵的栈mStack、栈顶索引stackTop。注意，笔者

开发的这个栈的最大深度是10，读者如果有需要可以进行修改。

- 第10-13行为产生不进行任何实际变换的初始变换矩阵的方法，后继的各种变换随着各种变换方法的调用都将叠加到此矩阵里。
- 第14-24行为将当前变换矩阵状态保存进栈的pushMatrix方法，以及将栈顶的变换矩阵恢复到当前变换矩阵中的popMatrix方法。这两个方法在复杂场景的开发中非常有用，读者在后继的学习中会慢慢地感觉到。
- 第25-27行为沿X、Y、Z轴方向进行平移变换的方法，该方法是通过调用Matrix类的translateM方法实现的。具体功能为将指定平移变换对应的平移变换矩阵乘以当前变换矩阵，再将最终结果矩阵存储到当前变换矩阵中。
- 第29-34行为根据当前变换矩阵、摄像机观察矩阵、投影矩阵生成最终变换矩阵的getFinalMatrix方法。其基本算法与升级前一致，只是变换矩阵不再由方法参数传入，而是采用当前变换矩阵。

> **提示**
>
> 开发中，使用此类时要首先调用setlnitStack方法对当前变换矩阵进行初始化，否则可能会造成程序不能正常执行。如本案例中，在渲染器的onSurfaceChanged方法中就调用了setlnitStack方法初始化了当前变换矩阵。

5.3.3 旋转变换

介绍旋转变换的矩阵格式前，首先需要了解一下绕坐标轴或任意轴旋转的一些规定。OpenGL ES中，旋转角度的正负可以用右手螺旋定则来确定，具体情况如图5-12所示。

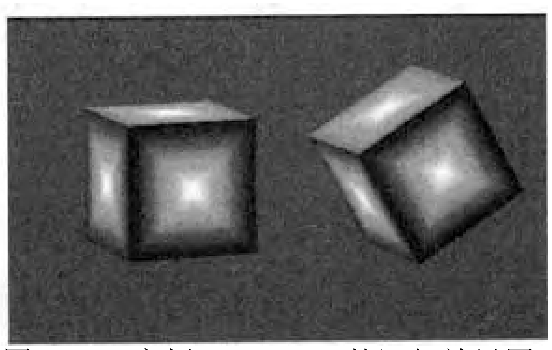


图5-12　案例Sample5_4的运行效果图

从图5-11中可以看出，所谓右手螺旋定则是指：右手握住旋转轴，使大姆指指向旋转轴的正方向，4指环绕的方向即为旋转的正方向，也就是旋转角度为正值。

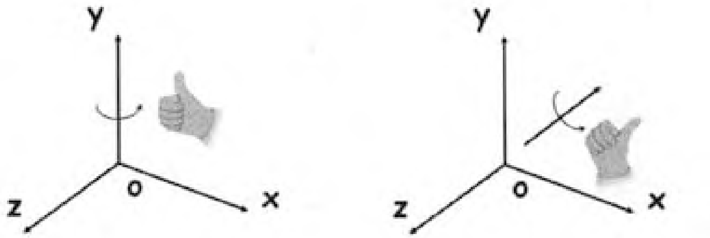


图5-11　旋转正方向的确定

旋转矩阵M的基本格式如下。

$$M=\begin{pmatrix} \cos\theta+(1-\cos\theta)u_x^2 & (1-\cos\theta)u_y u_x-\sin\theta u_z & (1-\cos\theta)u_z u_x+\sin\theta u_y & 0 \\ (1-\cos\theta)u_x u_y+\sin\theta u_z & \cos\theta+(1-\cos\theta)u_y^2 & (1-\cos\theta)u_z u_y-\sin\theta u_x & 0 \\ (1-\cos\theta)u_x u_z-\sin\theta u_y & (1-\cos\theta)u_y u_z+\sin\theta u_x & \cos\theta+(1-\cos\theta)u_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

上述矩阵表示将指定的点P绕轴向量u旋转θ度，其中的u_X 、u_Y 、u_Z 表示u向量在X、Y、Z 轴上的分量。由于本书不是专门讨论图形学相关数学的书籍，而旋转的计算比较复杂，这里就不进行验证计算了，有兴趣的读者可以参考其他资料进行计算验证。

了解了旋转变换矩阵的基本情况后，下面给出一个使用了旋转变换的简单案例——Sample5_4，其运行效果如图5-12所示。

了解了本案例的运行效果后，下面就可以进行代码的开发了。由于实际上本案例仅仅是将案例Sample5_3复制了一份并进行了修改，因此这里仅给出修改的步骤，如下所列。

（1）首先需要修改的是MySurfaceView类中的onDrawFrame方法，具体代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_4/com/bn/Sample5_4目录下的MySurfaceView.java。

```
1public void onDrawFrame(GL10 gl) {
```

```
GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT    //清除深度缓冲与颜色缓冲
        | GLES20.GL_COLOR_BUFFER_BIT);
//绘制原立方体
MatrixState.pushMatrix();                    //保护现场
cube.drawSelf();                             //绘制立方体
MatrixState.popMatrix();                     //恢复现场
//绘制变换后的立方体
MatrixState.pushMatrix();                    //保护现场
    MatrixState.translate(4, 0, 0);          //沿x方向平移4
    MatrixState.rotate(30, 0, 0, 1);         //绕z轴旋转30°
    cube.drawSelf();                         //绘制立方体
    MatrixState.popMatrix();                 //恢复现场
}
```

提示

此方法中主要是修改了绘制变换后立方体的代码，将原先的平移变换后增加了绕z轴旋转30°的旋转变换。实际的效果就是两种变换的叠加，先平移再旋转，因此图5-13的运行效果图中在原立方体的右侧看到了平移、旋转变换后的另一个立方体。

（2）完成了MySurfaceView类的修改后，接着需要对MatrixState类再次升级，为其增加实现旋转变换的rotate方法，具体代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_4/com/bn/Sample5_4目录下的MatrixState.java。

```
public static void rotate(float angle, float x, float y, float z) {
    Matrix.rotateM(currMatrix, 0, angle, x, y, z);
}
```

• 参数angle为需要旋转的角度，参数x、y、z为旋转轴对应向量的X、Y、Z分量。例如，4个参数的值为30、0、0、1表示绕Z轴旋转30°，4个参数的值为45、0.57735、0.57735、0.57735表示绕与向量[0.57735，0.57735，0.57735]方向相同的轴旋转45°。

• 此方法与前面小节中的translate方法类似，是通过调用Matrix类的rotateM方法实现的。具体功能为将指定旋转变换对应的变换矩阵乘以当前的变换矩阵，再将最终结果矩阵存储到当前变换矩阵中。

5.3.4 缩放变换

前面已经介绍过，OpenGL ES中的基本变换都是通过变换矩阵完成的，缩放变换自然也是如此，其变换矩阵的基本格式如下。

$$\mathbf{M}=\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

上述矩阵中的s_x、s_y、s_z分别表示缩放变换中的沿X、Y、Z轴方向的缩放率。通过简单的线性 代数计算即可验证，矩阵M乘以变换前P点的齐次坐标后确实得到了相当于将P点坐标沿X、Y、Z轴方向缩放s_x、s_y、s_z倍的结果，具体情况如下。

$$\mathrm{MP}=\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}=\begin{pmatrix} s_xP_x \\ s_yP_y \\ s_zP_z \\ 1 \end{pmatrix}，即\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix}=\begin{pmatrix} s_xP_x \\ s_yP_y \\ s_zP_z \\ 1 \end{pmatrix}$$

了解了缩放变换矩阵的基本情况后，下面给出一个使用了缩放变换的简单案例——Sample5_5，其运行效果如图5-13所示。

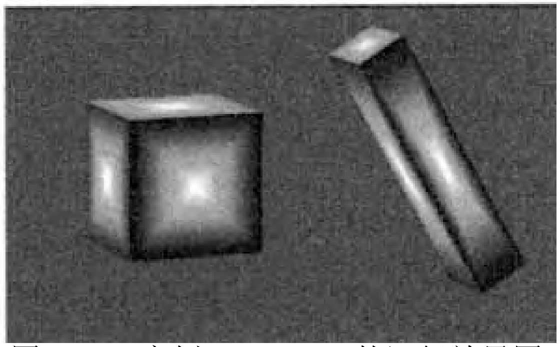


图5-13　案例Sample5_5的运行效果图

了解了本案例的运行效果后，下面就可以进行代码的开发了。由于实际上本案例仅仅是将案例Sample5_4复制了一份并进行了修改，因此这里仅给出修改的步骤，具体如下所列。

（1）首先需要修改的是MySurfaceView类中的onDrawFrame方法，具体代码如下。

代码位置：见随书光盘中源代码第5章/Sample5_4/com/bn/Sample5_4目录下的MySurfaceView.java。

```
//绘制变换后的立方体
MatrixState.pushMatrix();                    // 保护现场
MatrixState.translate(4, 0, 0);              // 沿 x 方向平移 4
MatrixState.rotate(30, 0, 0, 1);             // 绕 z 轴旋转 30°
MatrixState.scale(0.4f, 2f, 0.6f);           //x、y、z 3 个方向按各自的缩放因子进行缩放
cube.drawSelf();                             // 绘制立方体
MatrixState.popMatrix();                     // 恢复现场
```

提示

此方法中主要是修改了绘制变换后立方体的代码，在原先的旋转变换后增加了沿*X*、*Y*、*Z* 轴方向分别缩放0.4倍、2倍、0.6倍的缩放变换。实际的效果就是3种变换的叠加，先平移再旋转，最后缩放。因此图5-14的运行效果图中在原立方体的右侧看到了平移、旋转、缩放变换后的另一个立方体，此时实际已经变成长方体了。

（2）完成了MySurfaceView类的修改后，接着需要对MatrixState类进一步升级，为其增加实现缩放变换的scale方法，具体代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_5/com/bn/Sample5_5目录下的MatrixState.java。

```
public static void scale(float x, float y, float z) {    // 按x、y、z轴进行缩放
    Matrix.scaleM(currMatrix, 0, x, y, z);
}
```

> **提示**
>
> 与前面小节中的translate、rotate方法类似，该方法是通过调用Matrix类的scaleM方法实现的。具体功能为将指定缩放变换对应的缩放变换矩阵乘以当前变换矩阵，再将最终结果矩阵存储到当前变换矩阵中。

5.3.5　基本变换的实质

前面几个小节分别介绍了3种基本变换的实现方法，给读者的感觉应该是通过矩阵直接实现了对物体的变换。但在OpenGL ES中这并不完全准确，若仅仅这样来理解在非常简单的情况下可能没有问题，但在多个物体的组合变换中可能就会产生问题了。

这是因为变换实际上并不是直接针对物体的，而是针对坐标系进行的。OpenGL ES中变换的实现机制可以理解为首先通过矩阵对坐标系进行变换，然后根据传入渲染管线的原始顶点坐标在最终变换结果坐标系中的位置来进行绘制，图5-14基于案例Sample5_5的场景说明了这个问题。

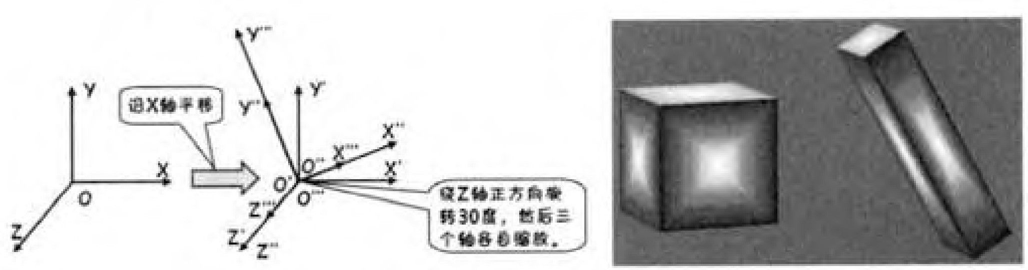


图5-14　基本变换的实质

从图5-14中可以看出。

- 左侧的是原始坐标系，原始坐标系首先向右沿*X* 轴进行了平移（得到上标为“’”的坐标系），然后绕Z轴旋转了30°（得到上标为“″”的坐标系），接着沿*X*、*Y*、*Z* 轴分别按不同的倍数缩放得到了最终的结果坐标系（上标为“″’”）。
- 最后渲染管线按照物体的原始顶点坐标值在最终结果坐标系里的位置进行绘制，这就得到了场景中右侧变换过的立方体。此时看起来已经变为斜着的长方体了，但对于其绘制坐标系而言其还是立方体，只是由于坐标系变化了才产生这种效果。

由于场景中可能有很多物体，这些物体都需要经过一系列变换后进行绘制，其中有些变换对于几个物体是共用的。这种情况下如果对每个物体都重新从原始坐标系开始进行变换就很烦琐，而且会进行一些不必要的重复计算。因此，系统中最好有保存变换矩阵状态的功能，这时前面笔者在MatrixState类中加入的pushMatrix与popMatrix方法就非常有用了。

考察这样的绘制需求，一共绘制3个立方体，第一个沿*X* 轴平移距离D后绘制，第二个在第一个的正上方距离D’的位置绘制，第三个在第一个的下方距离D″的位置绘制。实现方式有很多种，比较典型的两种如表5-1所列。

表5-1　两种不同的实现方法

操作序号	第一种方法	第二种方法
1	生成无变换原始坐标系	生成无变换原始坐标系
2	将坐标系沿*X* 轴平移距离D	将坐标系沿*X* 轴平移距离D
3	绘制第一个立方体	绘制第一个立方体
4	将坐标系沿F轴平移距离D’	pushMatrix保护现场
5	绘制第二个立方体	将坐标系沿*Y* 轴平移距离D’
6	生成无变换原始坐标系	绘制第二个立方体
7	将坐标系沿*X* 轴平移距离D	popMatrix恢复现场
8	将坐标系沿*Y* 轴平移距离D’’	将坐标系沿*Y* 轴平移距离D’
9	绘制第三个立方体	绘制第三个立方体

通过对表5-1中两种不同的实现方式进行比较可以看出，虽然步骤一样多，但左侧的方式进行变换的次数多一些，右侧的通过恰当的存储（保护现场）与取出（恢复现场）变换矩阵状态减少了变换次数，提高了效率。

提示

其实有了保护与恢复现场的功能后不但可以提高效率，而且在很多情况下可以将复杂问题简单化，在后面的学习中读者会慢慢体会到。

5.4 绘制方式

细心的读者会发现，本书前面所有的案例中最后绘制物体时都是采用的同一种方式——GL_TRIANGLES。其实OpenGL ES中支持的绘制方式还有很多，如GL_POINTS、GL_LINES等，本节将对各种绘制方式进行详细介绍。

5.4.1 各种绘制方式概览

OpenGL ES中支持的绘制方式大致分3类，包括点、线段、三角形，每类中包括一种或多种具体的绘制方式，各种具体绘制方式的说明如下所列。

- GL_POINTS方式。

此方式是点类下的唯一一个绘制方式了，其将传入渲染管线的一系列顶点单独进行绘制，不组装成更高一级的图元（如：线段、三角形等），具体情况如图5-15所示。

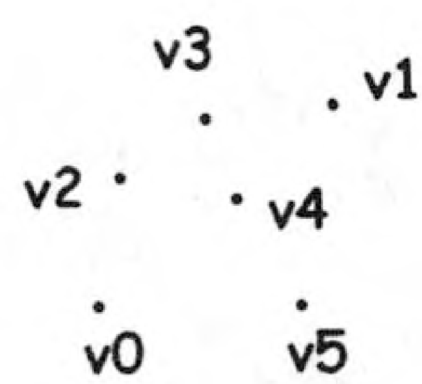


图5-15　GL_POINTS

- GL_LINES。

此方式是线段类的一种，其将传入渲染管线的一系列顶点按照顺序两两组织成线段进行绘制，具体情况如图5-16所示。要注意的是，若顶点个数为奇数，管线会自动忽略最后一个顶点。

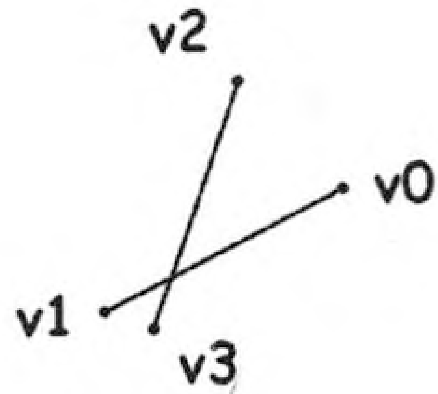


图5-16 GL_LINES

- GL_LINE_STRIP。

此方式也是线段类的一种，其将传入渲染管线的一系列顶点按照顺序依次组织成线段进行绘制，具体情况如图5-17所示。

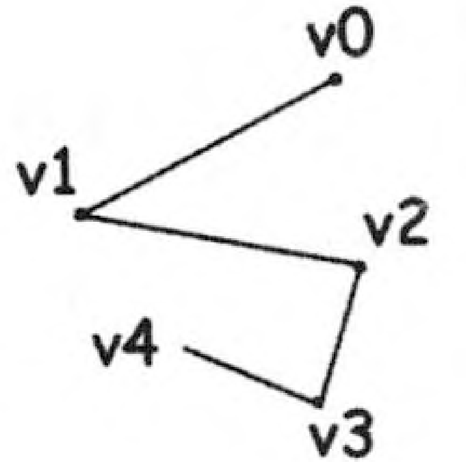


图5-17 GL_LINE_STRIP

- GL_LINE_LOOP。

此方式还是线段类的一种，其将传入渲染管线的一系列顶点按照顺序依次组织成线段进行绘制。与GL_LINE_STRIP方式的区别是，其将最后一个顶点与第一个顶点相连，形成线段环，具体情况如图5-18所示。

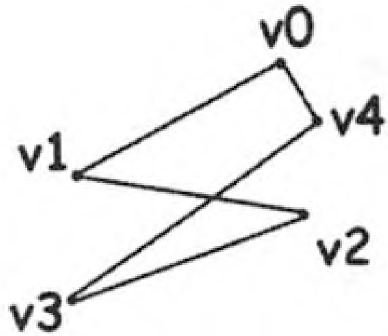


图5-18　GL_LINE_LOOP

提示

上述4幅图中顶点的编号代表的是顶点送入渲染管线顺序，后面关于其他绘制方式的示意图中也是如此。

- GL_TRIANGLES。

此方式是三角形类之一，其将传入渲染管线的一系列顶点按照顺序每3个组织成一个三角形进行绘制，具体情况如图5-19所示。从图5-19中可以看出，顶点2、5以及顶点1、3位置是相同的，这是因为左下侧和右上侧的两个三角形有共用顶点。

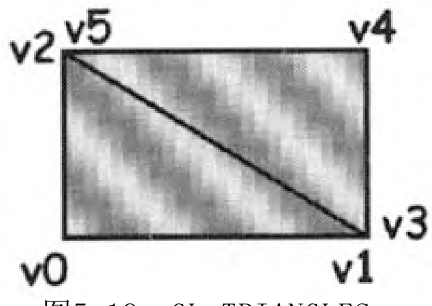


图5-19　GL_TRIANGLES

此绘制方式下，多个三角形有共用顶点时会造成冗余。如果不希望冗余可以用后面介绍的其他绘制方式完成绘制。

- GL_TRIANGLE_STRIP。

此方式也是三角形类之一，其将传入渲染管线的一系列顶点按照顺序依次组织成三角形进行绘制，最后实际形成的是一个三角形条带。若共有N

个顶点，则将绘制出N-2个三角形，具体情况如图5-20所示。通过比较可以发现，绘制同样的图形，此方式比GL_TRIANGLES方式节省顶点空间。

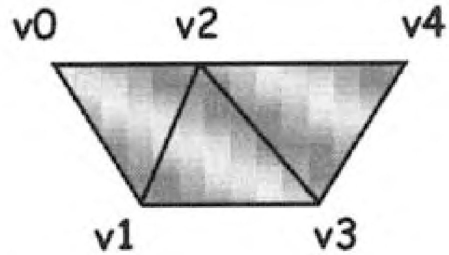


图5-20 GL_TRIANGLE_STRIP

- GL_TRIANGLE_FAN。

此方式还是三角形类之一，其将传入渲染管线的一系列顶点中的第一个顶点作为中心点，其他顶点作为边缘点绘制出一系列组成扇形的相邻三角形，具体情况如图5-21所示。

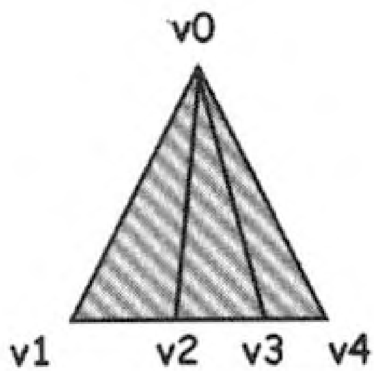


图5-21 GL_TRIANGLE_FAN

5.4.2 点与线段绘制方式

上一小节介绍了各种绘制方式的基本情况，本小节将通过一个简单的案例向读者详细介绍如何在开发中使用点及线段的绘制方式，其运行效果如图5-22所示。

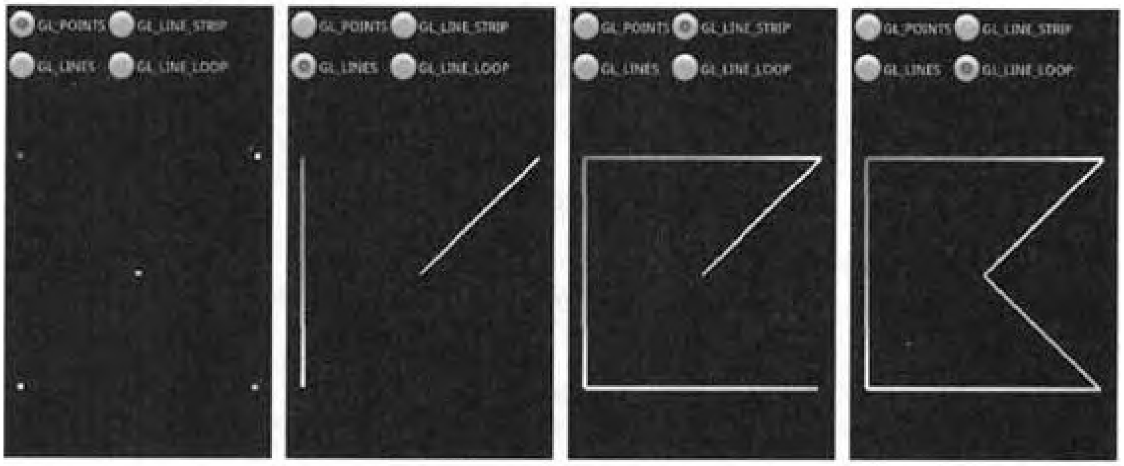


图5-22　案例Sample5_6的运行效果图

提示

图5-22中从左到右依次为用GL_POINTS、GL_LINES、GL_LINE_STRIP和GL_LINE_LOOP绘制方式绘制的效果图。

了解了本案例的运行效果后，就可以进行代码的开发了。由于本案例中大部分类的代码与前面案例中的非常相似，因此不再赘述，需要的读者请参考随书光盘中的源代码。下面仅介绍与前面很不同的采用点或线段绘制方式的PointsOrLines类，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_6/com/bn/Sample5_6目录下的PointsOrLines.java。

```
package com.bn.Sample5_6;                                    //声明包
import java.nio.ByteBuffer;                                  //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.opengl.GLES20;                                //引入相关类
public class PointsOrLines {                                 //颜色点或线
    int mProgram;                                            // 自定义渲染管线着色器程序 id
    ……//此处省略了部分成员变量，读者可自行查看随书光盘的源代码
    FloatBuffer mColorBuffer;                                // 顶点着色数据缓冲
    int vCount = 0;
        public PointsOrLines(MySurfaceView mv) {/*代码省略*/}
        public void initVertexData() {                       //初始化顶点坐标与着色数据的方法
            vCount = 5;
            float vertices[] = new float[] {
                    0, 0, 0, Constant.UNIT_SIZE, Constant.UNIT_SIZE, 0,
                    -Constant.UNIT_SIZE, Constant.UNIT_SIZE, 0,
                    -Constant.UNIT_SIZE, -Constant.UNIT_SIZE, 0,
                    Constant.UNIT_SIZE, -Constant.UNIT_SIZE, 0, };
            ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);
                                                             //创建顶点坐标数据缓冲
            vbb.order(ByteOrder.nativeOrder());              //设置字节顺序
            mVertexBuffer = vbb.asFloatBuffer();             //转换为 Float 型缓冲
            mVertexBuffer.put(vertices);                     //向缓冲区中放入顶点坐标数据
            mVertexBuffer.position(0);                       //设置缓冲区起始位置
            float colors[] = new float[] {         //顶点颜色值数组，每个顶点 4 个色彩值 RGBA
                    1, 1, 0, 0,                              //黄色
                    1, 1, 1, 0,                              //白色
                    0, 1, 0, 0,                              //绿色
                    1, 1, 1, 0,                              //白色
                    1, 1, 0, 0,                              //黄色
            };
            ByteBuffer cbb = ByteBuffer.allocateDirect(colors.length * 4);
                                                             //创建顶点着色数据缓冲
```

```
31          cbb.order(ByteOrder.nativeOrder());          //设置字节顺序
32          mColorBuffer = cbb.asFloatBuffer();          //转换为Float型缓冲
33          mColorBuffer.put(colors);                    //向缓冲区中放入顶点着色数据
34          mColorBuffer.position(0);                    //设置缓冲区起始位置
35      }
36      public void initShader(MySurfaceView mv){/*与前面案例类似，代码省略*/} //初始化着色器
37      public void drawSelf() {
38          GLES20.glUseProgram(mProgram);               //指定使用某套着色器程序
39          ……//此处省略了部分代码，读者可自行查看随书光盘的源代码
40          GLES20.glEnableVertexAttribArray(maColorHandle);//启用顶点着色数据
41          GLES20.glLineWidth(10);                      //设置线的宽度
42          switch (Constant.CURR_DRAW_MODE) {           //绘制点或线
43          case Constant.GL_POINTS:                     // GL_POINTS方式
44              GLES20.glDrawArrays(GLES20.GL_POINTS, 0, vCount);break;
45          case Constant.GL_LINES:                      // GL_LINES方式
46              GLES20.glDrawArrays(GLES20.GL_LINES, 0, vCount);break;
47          case Constant.GL_LINE_STRIP:                 // GL_LINE_STRIP方式
48              GLES20.glDrawArrays(GLES20.GL_LINE_STRIP, 0, vCount);break;
49          case Constant.GL_LINE_LOOP:                  // GL_LINE_LOOP方式
50              GLES20.glDrawArrays(GLES20.GL_LINE_LOOP, 0, vCount);break;
51  }}}
```

- 第11-35行为初始化顶点坐标与着色数据的方法，一共初始化了5个顶点的坐标与颜色，这5个顶点的位置及顺序如图5-23所示。

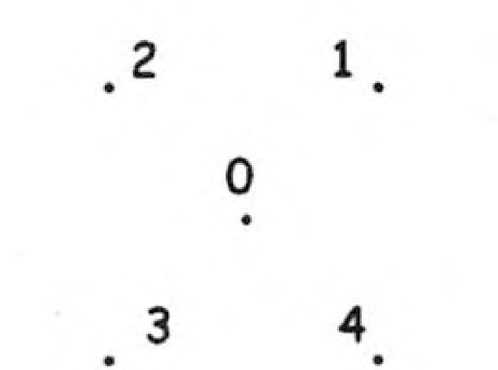


图5-23　顶点的位置与顺序

- 第42-50行根据当前选择的绘制方式编号，采用不同的绘制方式对前面初始化的5个顶点进行绘制。绘制方式的选择由用户通过界面上方的单选按钮进行，单选按钮相关事件的代码在Sample5_6_Activity类中，需要的读者可以参考随书光盘。

另外，本案例中用到的顶点与片元着色器与第三章Sample3_1中的基本一致，主要的变化就是在顶点着色器中加入了如下代码。

```
gl_PointSize=10.0;                    //设置点的尺寸
```

上述代码的功能为设置点绘制方式下点的大小，若没有进行设置，则管线采用默认值1。

5.4.3　三角形条带与扇面绘制方式

掌握了点与线段的绘制方式后，本小节将通过一个简单的案例向读者详细介绍如何在开发中使用三角形条带及扇面的绘制方式，其运行效果如图5-24所示。

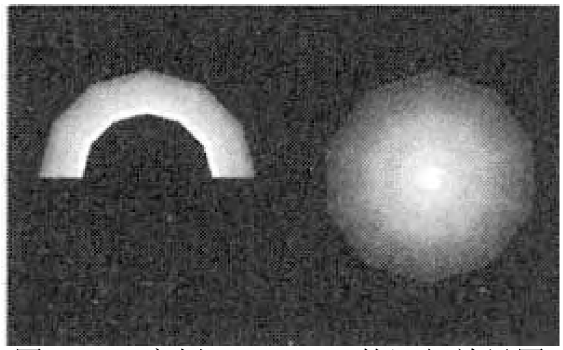


图5-24　案例Sample5_7的运行效果图

了解了本案例的运行效果后，就可以进行代码的开发了。由于本案例中大部分类的代码与前面案例中的非常相似，因此不再赘述，需要的读者请参考随书光盘中的源代码。下面的步骤中仅介绍与前面案例大不相同的使用三角形条带及扇面绘制方式的几个类的开发，具体如下所列。

（1）首先开发的是使用三角形条带（GL_TRIANGLE_STRIP）方式进行绘制的Belt类，其代码如下。



代码位置：见随书光盘中源代码/第5章/Sample5_7/com/bn/Sample5_7目录下的Belt.java。

```
package com.bn.Sample5_7;                                   //声明包
import java.nio.ByteBuffer;                                 //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.opengl.GLES20;                               //引入相关类
public class Belt {
    int mProgram;                                           //自定义渲染管线着色器程序 id
    ……//此处省略了部分成员变量，读者可自行查看随书光盘的源代码
    FloatBuffer mColorBuffer;                               //顶点着色数据缓冲
    int vCount = 0;                                         //顶点数量
        public Belt(MySurfaceView mv) {/*代码省略，需要的参考光盘*/}
        public void initVertexData() {                      //初始化顶点坐标与着色数据的方法
            int n = 6;
            vCount = 2 * (n + 1);
            float angdegBegin = -90;
            float angdegEnd = 90;
            float angdegSpan = (angdegEnd - angdegBegin) / n;
            float[] vertices = new float[vCount * 3];   //顶点坐标数据数组
            int count = 0;
            for (float angdeg = angdegBegin; angdeg <= angdegEnd; angdeg += angdegSpan)
            {
                double angrad = Math.toRadians(angdeg); //当前弧度
                vertices[count++] = (                   //顶点 x 坐标
                        float) (-0.6f * Constant.UNIT_SIZE * Math.sin(angrad));
                vertices[count++] = (                   //顶点 y 坐标
                        float) (0.6f * Constant.UNIT_SIZE * Math.cos(angrad));
                vertices[count++] = 0;                  //顶点 z 坐标
                vertices[count++] = (float) (-Constant.UNIT_SIZE * Math.sin(angrad));
                                                        //顶点 x 坐标
                vertices[count++] = (float) (Constant.UNIT_SIZE * Math.cos(angrad));
                                                        //顶点 y 坐标
                vertices[count++] = 0;                  //顶点 z 坐标
            }
            ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);
                                                        // 创建顶点坐标数据缓冲
            vbb.order(ByteOrder.nativeOrder());         // 设置字节顺序
            mVertexBuffer = vbb.asFloatBuffer();        // 转换为 Float 型缓冲
            mVertexBuffer.put(vertices);                // 向缓冲区中放入顶点坐标数据
            mVertexBuffer.position(0);                  // 设置缓冲区起始位置
            count = 0;
            float colors[] = new float[vCount * 4];
```

```
                                     //顶点颜色值数组，每个顶点 4 个色彩值 RGBA
37          for (int i = 0; i < colors.length; i += 8) {
38              colors[count++] = 1;colors[count++] = 1;colors[count++] = 1;colors
                [count++] = 0;
39              colors[count++] = 0;colors[count++] = 1;colors[count++] = 1;colors
                [count++] = 0;
40          }
41          ByteBuffer cbb = ByteBuffer.allocateDirect(colors.length * 4);
                                                    //创建顶点着色数据缓冲
42          cbb.order(ByteOrder.nativeOrder());     //设置字节顺序
43          mColorBuffer = cbb.asFloatBuffer();     //转换为 Float 型缓冲
44          mColorBuffer.put(colors);               //向缓冲区中放入顶点着色数据
45          mColorBuffer.position(0);               //设置缓冲区起始位置
46      }
47      public void initShader(MySurfaceView mv) {/*代码省略*/}  //初始化着色器
48      public void drawSelf() {
49          GLES20.glUseProgram(mProgram);              //指定使用某套着色器程序
50          ……//此处省略了部分代码，读者可自行查看随书光盘的源代码
51          GLES20.glEnableVertexAttribArray(maColorHandle);//启用顶点颜色数据
52          GLES20.glDrawArrays(GLES20.GL_TRIANGLE_STRIP, 0, vCount);
                                                    //采用三角形条带方式绘制
53  }}
```

• 第11-46行为初始化三角形条带中各个顶点坐标与着色数据的方法，其中各顶点的位置及顺序如图5-25所示。

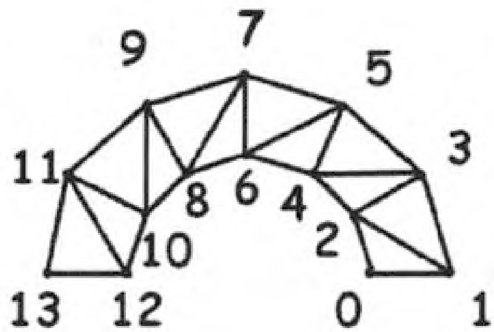


图5-25　条带中各顶点的顺序

- 第48-53行为采用三角形条带方式绘制物体的方法。

（2）完成了采用三角形条带方式进行绘制的Belt类的开发后，下面将要开发的是采用三角形扇面方式进行绘制的Circle类，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_7/com/bn/Sample5_7目录下的Circle.java。

```
package com.bn.Sample5_7;                                     //声明包
import java.nio.ByteBuffer;                                   //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.opengl.GLES20;                                 //引入相关类
public class Circle {                                         //颜色圆
    int mProgram;                                             //自定义渲染管线着色器程序id
    ……//此处省略了部分成员变量的声明，读者可自行查看随书光盘中的源代码
    FloatBuffer mColorBuffer;                                 //顶点着色数据缓冲
    int vCount = 0;
        public Circle(MySurfaceView mv) {/*代码省略*/}
        public void initVertexData() {                        //初始化顶点坐标与着色数据的方法
            int n = 10;
            vCount = n + 2;
            float angdegSpan = 360.0f / n;
            float[] vertices = new float[vCount * 3];         //顶点坐标数据数组
            int count = 0;
            vertices[count++] = 0;vertices[count++] = 0;vertices[count++] = 0;
                                                              //第一个顶点的坐标
```

```
18          for (float angdeg = 0; Math.ceil(angdeg) <= 360; //循环生成其他顶点的坐标
19          angdeg += angdegSpan) {
20              double angrad = Math.toRadians(angdeg);     //当前弧度
21              vertices[count++] = (float) (-Constant.UNIT_SIZE * Math.sin(angrad));
                                                            //顶点 X 坐标
22              vertices[count++] = (float) (Constant.UNIT_SIZE * Math.cos(angrad));
                                                            //顶点 Y 坐标
23              vertices[count++] = 0;                      //顶点 Z 坐标
24          }
25          ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);
                                                            //创建顶点坐标数据缓冲
26          vbb.order(ByteOrder.nativeOrder());             //设置字节顺序
27          mVertexBuffer = vbb.asFloatBuffer();            //转换为 Float 型缓冲
28          mVertexBuffer.put(vertices);                //向缓冲区中放入顶点坐标数据
29          mVertexBuffer.position(0);                      //设置缓冲区起始位置
30          count = 0;
31          float colors[] = new float[vCount * 4];
                                          //顶点颜色值数组，每个顶点 4 个色彩通道 RGBA
32          colors[count++] = 1;colors[count++] = 1;        //第一个顶点的颜色:白色
33          colors[count++] = 1;colors[count++] = 0;
34          for (int i = 4; i < colors.length; i += 4) {    //剩余顶点的颜色:绿色
35              colors[count++] = 0;colors[count++] = 1;colors[count++] = 0;colors
                [count++] = 0;
36          }
37          ByteBuffer cbb = ByteBuffer.allocateDirect(colors.length * 4);
                                                            //创建顶点着色数据缓冲
38          cbb.order(ByteOrder.nativeOrder());             //设置字节顺序
39          mColorBuffer = cbb.asFloatBuffer();             //转换为 Float 型缓冲
40          mColorBuffer.put(colors);                   //向缓冲区中放入顶点着色数据
41          mColorBuffer.position(0);                       //设置缓冲区起始位置
42      }
43      public void initShader(MySurfaceView mv){/*代码省略*/}//初始化着色器
44      public void drawSelf() {
45          GLES20.glUseProgram(mProgram);              //指定使用某套着色器程序
46          ……//此处省略了部分代码，读者可自行查看随书光盘的源代码
47          GLES20.glEnableVertexAttribArray(maColorHandle);
48          GLES20.glDrawArrays(GLES20.GL_TRIANGLE_FAN, 0, vCount);
                                                    //采用三角形扇面方式绘制
49  }}
```

- 第11-42行为初始化三角形扇面中各个顶点坐标与着色数据的方法，其中各顶点的位置及顺序如图5-26所示。

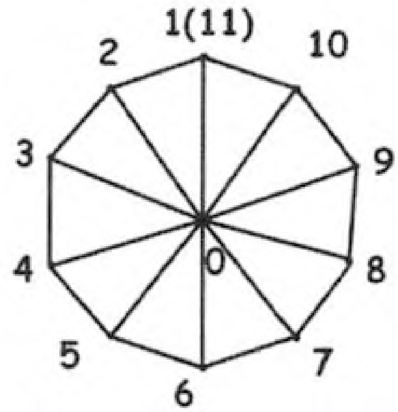


图5-26　扇面中各顶点的顺序

- 第44-49行为采用三角形扇面方式绘制物体的方法。

从上述案例中可以看出，采用三角形条带或扇面方式绘制物体时需要的顶点数量一般情况下会比直接的三角形方式（GL_TRIANGLES）少，因此在情况允许时应该尽量采用三角形条带或扇面方式。这是因为渲染管线在执行时对传入管线的每个顶点都将执行一次顶点着色器，需要的顶点越少绘制同样图形的速度就越快。

读者可能会觉得三角形条带方式虽然效率高，但只适合用来绘制连续三角形构成的物体，若是 非连续三角形构成的物体就没有用武之地了。其实不然，非连续三角形构成的物体一样可以使用三角形条带方式进行绘制，图5-27说明了这个问题。

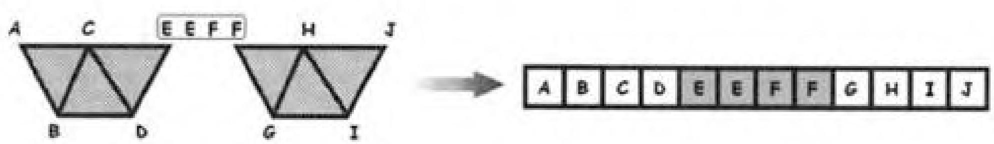


图5-27　用三角形条带方式绘制非连续三角形构成的物体

从图5-27中可以看出，当要使用三角形条带方式绘制非连续三角形构成的物体时，只要将上一批三角形的最后一个顶点和下一批三角形的第一个顶点重复一次即可。如图5-27中要绘制由顶点A～J构成的6个三角形，送入渲染管线的顶点序列为“ABCDEEFFGHIJ”，这比直接三角形方式还是节省了不少顶点，有助于渲染效率的提高。

了解了采用三角形条带方式绘制由非连续三角形构成物体的原理后，下面将给出一个采用了此技术的案例——Sample5_8，其运行效果如图5-28所示。

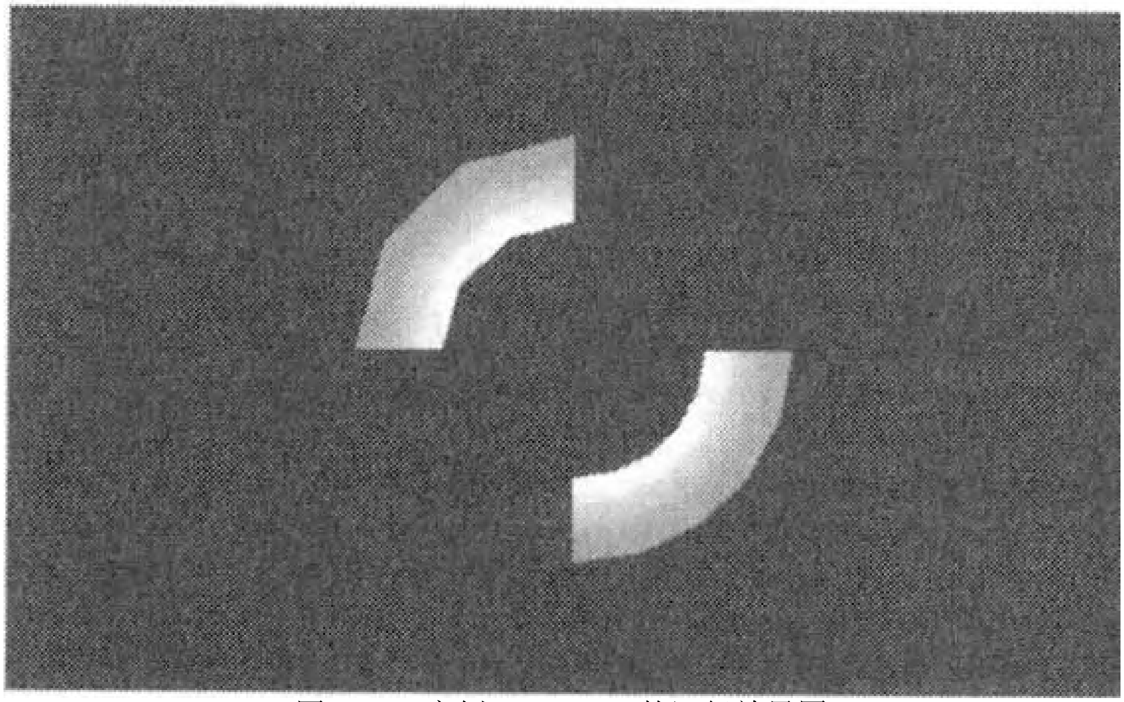


图5-28　案例Sample5_8的运行效果图

其实本案例只是将前面的案例Sample5_7复制了一份，去掉了里面绘制扇面的代码，修改了采用三角形条带绘制方式的Belt类。因此这里仅介绍修改后的Belt类，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_8/com/bn/Sample5_8目录下的Belt.java。

```
package com.bn.Sample5_8;                                    //声明包
import java.nio.ByteBuffer;                                  //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.opengl.GLES20;                                //引入相关类
public class Belt {                                          //颜色条状物
    int mProgram;                                            //自定义渲染管线着色器程序 id
    ……//此处省略了部分成员变量，读者可自行查看随书光盘的源代码
    FloatBuffer mColorBuffer;                                //顶点着色数据缓冲
    int vCount = 0;
        public Belt(MySurfaceView mv) {/*代码省略*/}
        public void initVertexData() {                       //初始化顶点坐标与着色数据的方法
            int n1 = 3;
            int n2 = 5;
            vCount = 2 * (n1 + n2 + 2) + 2;                  //计算总顶点数
            float angdegBegin1 = 0;
```

```
        float angdegEnd1 = 90;
        float angdegSpan1 = (angdegEnd1 - angdegBegin1) / n1;
        float angdegBegin2 = 180;
        float angdegEnd2 = 270;
        float angdegSpan2 = (angdegEnd2 - angdegBegin2) / n2;
        float[] vertices = new float[vCount * 3];        //顶点坐标数据
        int count = 0;
       for (float angdeg = angdegBegin1; angdeg <= angdegEnd1; angdeg += angdegSpan1)
{
            double angrad = Math.toRadians(angdeg);      //当前弧度
            vertices[count++] = (                        //大圆上的点
                    float) (-0.6f * Constant.UNIT_SIZE * Math.sin(angrad));
                                                         // 顶点坐标
            vertices[count++] = (
                    float) (0.6f * Constant.UNIT_SIZE * Math.cos(angrad));
            vertices[count++] = 0;
            vertices[count++] = (float) (-Constant.UNIT_SIZE * Math.sin(angrad));
                                                         // 小圆上的点
            vertices[count++] = (float) (Constant.UNIT_SIZE * Math.cos(angrad));
            vertices[count++] = 0;
        }
        vertices[count++] = vertices[count - 4];    //重复第一批三角形的最后一个顶点
        vertices[count++] = vertices[count - 4];
        vertices[count++] = 0;
       for (float angdeg = angdegBegin2; angdeg <= angdegEnd2; angdeg += angdegSpan2)
{
            double angrad = Math.toRadians(angdeg); //当前弧度
            if (angdeg == angdegBegin2) {           //重复第二批三角形的第一个顶点
                vertices[count++] = (float) (-0.6f * Constant.UNIT_SIZE * Math
                        .sin(angrad));              //顶点坐标
                vertices[count++] = (float) (0.6f * Constant.UNIT_SIZE * Math
                        .cos(angrad));
                vertices[count++] = 0;
            }
            vertices[count++] = (                   //大圆上的点
                    float) (-0.6f * Constant.UNIT_SIZE * Math.sin(angrad));
            vertices[count++] = (float) (0.6f * Constant.UNIT_SIZE * Math
                    .cos(angrad));
            vertices[count++] = 0;
            vertices[count++] = (                   //小圆上的点
                    float) (-Constant.UNIT_SIZE * Math.sin(angrad));
            vertices[count++] = (float) (Constant.UNIT_SIZE * Math.cos(angrad));
            vertices[count++] = 0;
        }
        ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);
                                                    //创建顶点坐标数据缓冲
        vbb.order(ByteOrder.nativeOrder());         //设置字节顺序
        mVertexBuffer = vbb.asFloatBuffer();        //转换为 Float 型缓冲
        mVertexBuffer.put(vertices);                //向缓冲区中放入顶点坐标数据
        mVertexBuffer.position(0);                  //设置缓冲区起始位置
        ……//此处省略了顶点颜色缓冲数据的初始化代码，读者可自行查看随书光盘的源代码
    }
```

```
    public void initShader(MySurfaceView mv) {/*与原来相同，代码省略*/} //初始化着色器
    public void drawSelf(){ /*与原来相同，代码省略*/}
}
```

> **提示**
>
> 从上述代码中可以看出，相比于修改前，主要是改变了顶点的位置。物体不再是整体由一批连续三角形组成，而是由两批连续三角形组成。第一批的最后一个顶点和第二批的第一个顶点在顶点序列中重复了，各顶点的位置及顺序如图5-29所示。

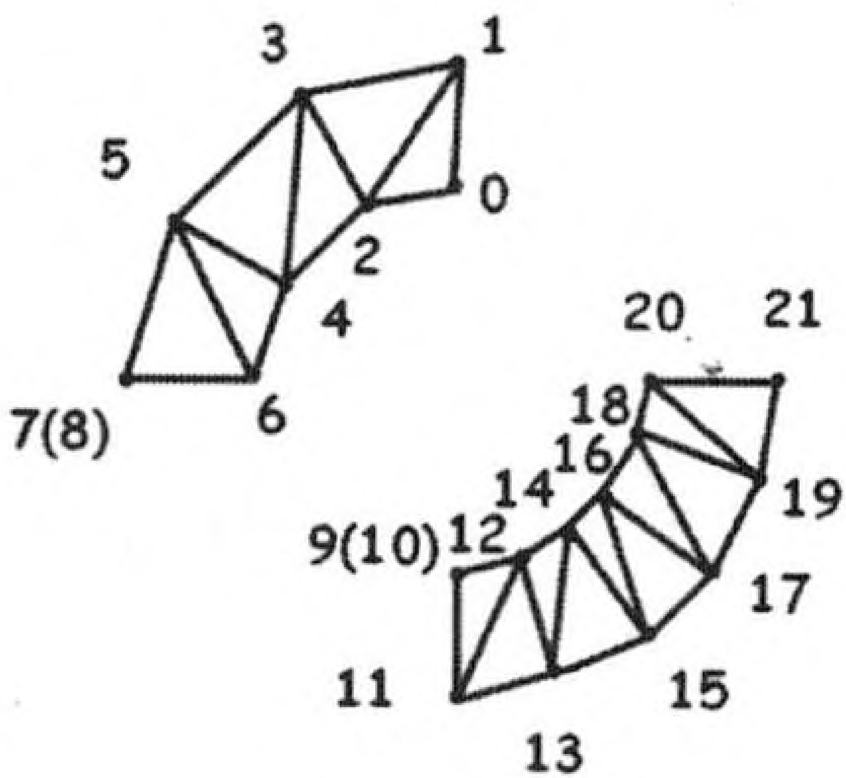


图5-29　Belt类中各顶点的顺序

5.4.4　顶点法与索引法

前面章节的案例中都是调用的glDrawArrays方法来进行物体的绘制，此方法是按照传入渲染管线顶点本身的顺序及选用的绘制方式将顶点组织成图元进行绘制的，也可以称之为顶点法。采用这种方法虽然很方便，但有些情况下不得不在顶点序列中出现很多重复的顶点，若希望减少重复顶点占用的空间，可以考虑采用glDrawElements方法来进行物体的绘制。

glDrawElements方法在绘制时不但要将顶点序列传入渲染管线，还需要将索引序列传入管线。绘制时管线根据索引值序列中的索引一一从顶点序列中取出对应的顶点，并根据当前选用的绘制方式组织成图元进行绘制，图5-30很好地说明了这个问题。

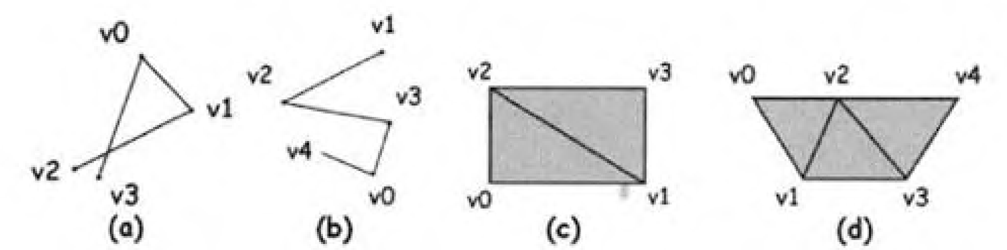


图5-30　采用索引法进行绘制

图5-30中给出了用索引法进行绘制时的示意图，具体含义如下所列。

- 图a表示的是采用GL_LINES方式进行绘制，顶点序列为{v0，v1，v2，v3}，索引序列为{0，1，1，2，0，3}。
- 图b表示的是采用GL_LINE_STRIP方式进行绘制，顶点序列为{v0，v1，v2，v3，v4}，索引序列为{1，2，3，0，4}。
- 图c表示的是采用GL_TRIANGLES方式进行绘制，顶点序列为{v0，v1，v2，v3}，索引序列为{0，1，2，1，3，2}。
- 图d表示的是采用GL_TRIANGLE_STRIP方式进行绘制，顶点序列为{v0，v1，v2，v3，v4}，索引序列为{0，1，2，3，4}。

提示

从上述情况中可以看出，采用索引法（glDrawElements）进行绘制时可以有效地减少重复顶点数据，有重复时只需要提供重复的索引号即可。而每个索引值只需要一个整数，比顶点值需要3个整数或浮点数可以节省不少空间。上述内容中没有给出的GL_POINTS、GL_LINE_LOOP、GL_TRIANGLE_FAN等3种方式也支持索引法，依此类推即可。

了解了索引法绘制的基本原理后，下面给出一个采用索引法进行绘制的简单案例——Sample5_9。其实本案例仅仅是将前面的Sample5_7中的绘制方法改为了glDrawElements，并相应地修改了顶点序列，同时给出了索引序列。因此，这里仅仅介绍改动后的Circle类，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_9/com/bn/Sample5_9目录下的Circle.java。

```
package com.bn.Sample5_9;                                          //声明包
import java.nio.ByteBuffer;                                        //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.opengl.GLES20;                                      //引入相关类
public class Circle {
    int mProgram;                                                  //自定义渲染管线着色器程序 id
    ……//此处省略了部分成员变量的声明，读者可自行查看随书光盘中的源代码
    FloatBuffer mVertexBuffer;                                     //顶点坐标数据缓冲
    FloatBuffer mColorBuffer;                                      //顶点着色数据缓冲
        private ByteBuffer mIndexBuffer;                           //顶点索引数据缓冲
        int vCount = 0;                                            //顶点数量
        int iCount = 0;                                            //索引数量
        public Circle(MySurfaceView mv) {/*代码省略*/}
        public void initVertexData() {                             //初始化顶点数据与着色数据的方法
            int n = 10;
            vCount = n + 2;
            float angdegSpan = 360.0f / n;
            float[] vertices = new float[vCount * 3];              //顶点坐标数据
            int count = 0;
            vertices[count++] = 0;vertices[count++] = 0;vertices[count++] = 0;
                                                                   //顶点 x、y、z 坐标
            for (float angdeg = 0; Math.ceil(angdeg) <= 360; angdeg += angdegSpan) {
                double angrad = Math.toRadians(angdeg);            //当前弧度
                vertices[count++] = (float) (-Constant.UNIT_SIZE * Math.sin(angrad));
                                                                   //顶点 x 坐标
                vertices[count++] = (float) (Constant.UNIT_SIZE * Math.cos(angrad));
                                                                   //顶点 y 坐标
                vertices[count++] = 0;                             //顶点 z 坐标
            }
            ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);
                                                                   //创建顶点坐标数据缓冲
            vbb.order(ByteOrder.nativeOrder());                    //设置字节顺序
            mVertexBuffer = vbb.asFloatBuffer();                   //转换为 Float 型缓冲
            mVertexBuffer.put(vertices);                           //向缓冲区中放入顶点坐标数据
            mVertexBuffer.position(0);                             //设置缓冲区起始位置
            iCount = vCount;
            byte indices[] = new byte[iCount];                     //生成索引序列
            for(int i=0; i<iCount; i++){
                indices[i] = (byte) i;
            }
            mIndexBuffer = ByteBuffer.allocateDirect(indices.length);
                                                               //创建三角形构造索引数据缓冲
            mIndexBuffer.put(indices);                         //向缓冲区中放入三角形构造索引数据
```

```
39            mIndexBuffer.position(0);                    //设置缓冲区起始位置
40            ……//此处省略了顶点颜色缓冲数据的初始化代码，读者可自行查看随书光盘的源代码
41        }
42        public void initShader(MySurfaceView mv) {/*代码省略*/}    //初始化着色器
43        public void drawSelf() {                          //绘制圆
44            GLES20.glUseProgram(mProgram);               //指定使用某套着色器程序
45            ……//此处省略了部分代码，读者可自行查看随书光盘的源代码
46            GLES20.glEnableVertexAttribArray(maColorHandle);//启用着色数据
47            GLES20.glDrawElements(GLES20.GL_TRIANGLE_FAN, iCount,//用索引法绘制图形
48                    GLES20.GL_UNSIGNED_BYTE, mIndexBuffer);
49    }}
```

- 第6-12行为该类一些成员变量的声明，主要是增加了索引数据缓冲mIndexBuffer及索引数量iCount。
- 第14-41行为初始化顶点数据的initVertexData方法，其中除了对顶点的位置、着色数据进行了初始化外，还增加了顶点索引数据初始化的代码。
- 第43-49行为执行绘制的方法，其中第47行中调用的是GLES20类的glDrawElements方法根据索引绘制图形，而不再是前面案例使用的glDrawArrays方法。

> **提示**
>
> 此案例中绘制条带的Belt类也需要进行相应地修改，修改思路基本与Circle类一致，这里不再赘述，需要的读者可以参考随书光盘中的源代码。

5.5　设置合理的视角

使用过照相机的读者都知道，拍摄时根据不同的情况应当选用不同焦距的镜头，这些不同焦距镜头的一大区别就是视角不同。在同样的位置，

视角大可以观察到更宽范围内的景物，但投影到照片里的景物较小；视角小可以观察到的景物范围就窄一些，但投影到照片里的景物也就大一些。

同样，在OpenGL ES的虚拟世界中，摄像机也有视角大小的问题，具体情况如图5-31所示。

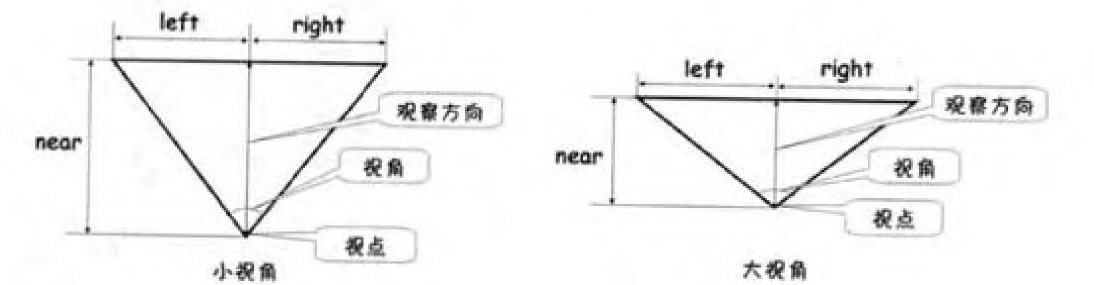


图5-31　小视角和大视角

提示

从图5-31中可以看出，OpenGL ES虚拟世界摄像机的左右视角大小主要是由left、right及near值决定的。可以想象出上下视角则由top、bottom及near值决定。这些值（left、right、top、bottom、near）都是生成投影矩阵时需要提供的参数。

视角的大小可以根据公式计算出来，例如水平方向视角的计算公式如下：

$$\alpha = 2\mathrm{arctg}(left/near)$$

而垂直方向的视角计算公式为：

$$\alpha = 2\mathrm{arctg}(top/near)$$

提示

上述两个计算公式是在左右或上下对称的（即left等于right，top等于bottom）情况下推导出的，若左右或上下不对称，则两个半角要分别计算。

同时，还可以从上述介绍中总结出如下规律。

- 在left、right、top、bottom值不变的情况下，near值越小，视角越大，反之则视角越小。

• 在near值不变的情况下，left、right、top、bottom值越大，视角越大，反之则视角越小。

因此，开发中当希望观察到同样范围的场景时就不止一种选择：可以将摄像机离得近一些，同时将视角设置得大一些；也可以将摄像机离得远一些，同时将视角设置得小一些；具体情况如图5-32所示。

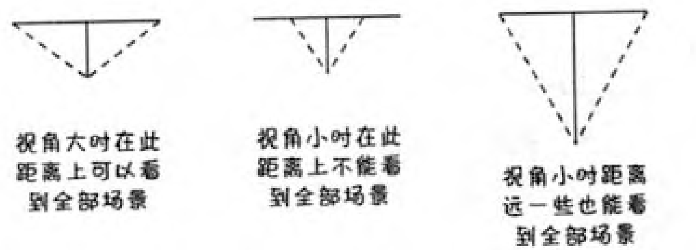


图5-32　不同视角观察同样范围的场景

经过上面的介绍读者可能会觉得，既然有多种选择，开发时随意选择一种能观察到指定范围的组合即可。但事实不完全是这样，因为观察时不但有看不看得到的问题，还有变不变型的问题。当视角很大时，往往有比较严重的变形，就像照相机的鱼眼镜头一样。因此开发中要选择合理的组合，使得既能观察到指定范围的场景，又能满足对物体不变型的需要。

提示

在某些特定的应用程序中，就是需要出现变形的效果，这时采用哪些可以出现合适变形效果的组合即可。

前面介绍了不同视角观察场景的基本原理，下面给出一个用不同视角在不同距离上观察同样范围场景的案例——Sample5_10，其运行效果如图5-33所示。

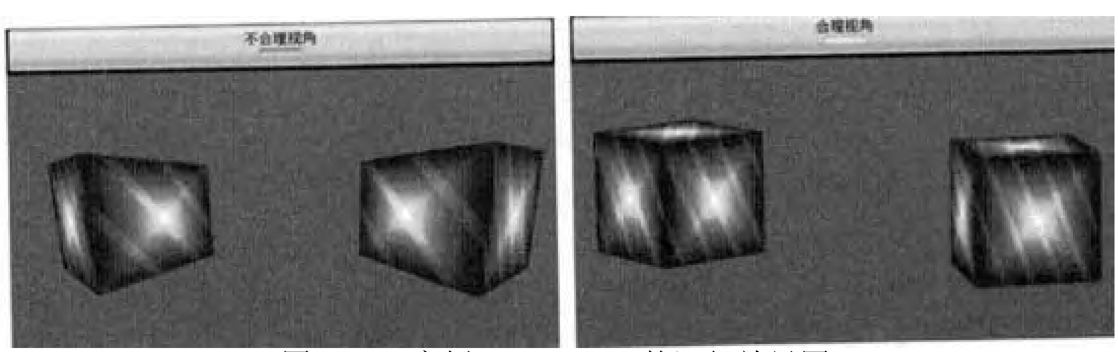


图5-33　案例Sample5_10的运行效果图

说明

左侧的效果图是摄像机离得近时采用大视角的观察情况，右侧的是摄像机离得远时采用小视角的观察情况。从左右两幅图的对比中可以看出，虽然在不同的距离采用不同的视角可以观察到基本相同范围内的场景，但大视角情况下变形较为严重，根本看不出来是两个立方体，但小视角时变形的情况就几乎觉察不到了。

了解了本案例的运行效果后，下面对本案例的具体开发过程进行详细介绍，具体步骤如下所列。

（1）首先需要开发的是本案例的主控制类Sample5_10_Activity，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_10/com/bn/Sample5_10目录下的Sample5_10_Activity.java。

```java
package com.bn.Sample5_10;                                  //声明包
import android.app.Activity;                                //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import static com.bn.Sample5_10.Constant.*;                 //引入相关类
public class Sample5_10_Activity extends Activity {
    private MySurfaceView mGLSurfaceView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ……//省略了部分与前面案例基本相同的代码，需要时请查看随书光盘
            //为选择不同视角组合的ToggleButton添加监听器
            ToggleButton tb = (ToggleButton) this.findViewById(R.id.ToggleButton01);
            tb.setOnCheckedChangeListener(new OnCheckedChangeListener() {
                public void onCheckedChanged(CompoundButton buttonView,
                        boolean isChecked) {
                    if (isChecked) {                        //视角不合适导致变形的情况
                        MatrixState.setProjectFrustum( //调用此方法计算产生透视投影矩阵
                                -ratio * 0.7f, ratio * 0.7f, -0.7f, 0.7f, 1, 10);
                        MatrixState.setCamera(          //调用此方法产生摄像机观察矩阵
                                0, 0.5f, 4, 0f, 0f, 0f, 0f, 1.0f, 0.0f);
                    } else {                                //视角合适不变形的情况
                        MatrixState.setProjectFrustum( //调用此方法计算产生透视投影矩阵
                                -ratio, ratio, -1, 1, 20, 100);
                        MatrixState.setCamera(          //调用此方法产生摄像机观察矩阵
                                0, 8f, 45, 0f, 0f, 0f, 0f, 1.0f, 0.0f);
                    }}});
        }
        ……//此处省略了onResume和onPause方法的代码，读者可自行查看随书光盘的源代码
    }
```

说明

第11-25行为选择不同视角/距离组合的开关按钮（ToggleButton）添加了监听器。监听器的事件处理方法中根据开关按钮的选中状态设置不同的摄像机位置及投影参数组合。

（2）完成了Sample5_10_Activity类的开发后，就可以开发用于渲染3D场景的类——MySurfaceView了，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_10/com/bn/Sample5_10目录下的MySurfaceView.java。

```
package com.bn.Sample5_10;                                  //声明包
import android.opengl.GLSurfaceView;                        //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import static com.bn.Sample5_10.Constant.*;                 //引入相关类
```

```
class MySurfaceView extends GLSurfaceView {
    private final float TOUCH_SCALE_FACTOR = 180.0f / 320;//角度缩放比例
    private SceneRenderer mRenderer;                    //场景渲染器
    private float mPreviousX;                           //上次的触控位置 X 坐标
        float yAngle = 0;                               //总场景绕 y 轴旋转的角度
        ……//此处省略了构造器和 onTouchEvent 方法的代码，读者可自行查看随书光盘的源代码
        private class SceneRenderer implements GLSurfaceView.Renderer {
            Cube cube;                                  //立方体对象引用
            public void onDrawFrame(GL10 gl) {
                GLES20.glClear(                         //清除深度缓冲与颜色缓冲
                        GLES20.GL_DEPTH_BUFFER_BIT| LES20.GL_COLOR_BUFFER_BIT);
                MatrixState.pushMatrix();               //保护现场
                MatrixState.rotate(yAngle, 0, 1, 0);    //绕 Y 轴旋转（实现总场景旋转）
                //平移并旋转后绘制左侧立方体
                MatrixState.pushMatrix();               //保护现场
                MatrixState.translate(-3, 0, 0);        //平移
                MatrixState.rotate(60, 0, 1, 0);        //旋转
                cube.drawSelf();
                MatrixState.popMatrix();                //恢复现场
                ///平移并旋转后绘制右侧立方体
                MatrixState.pushMatrix();               //保护现场
                MatrixState.translate(3, 0, 0);         //平移
                MatrixState.rotate(-60, 0, 1, 0);       //旋转
                cube.drawSelf();
                MatrixState.popMatrix();                //恢复现场
                MatrixState.popMatrix();                //恢复现场
            }
    ……//省略了与前面案例代码类似的 onSurfaceChanged 及 onSurfaceCreated 方法代码
    }}
```

说明

第14-32行为用于绘制整个场景的onDrawFrame方法，其中首先根据yAngle值（此值可以通过用户手指触动屏幕改变）将坐标系绕*Y*轴进行旋转变换，然后分别基于此旋转后的坐标系再进行两组独立的平移、旋转变换绘制了场景中左右两侧的立方体。

（3）完成了MySurfaceView类的开发后，就可以开发本案例中用于绘制立方体的Cube类了，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_10/com/bn/Sample5_10目录下的Cube.java。

```
package com.bn.Sample5_10;                          //声明包
import static com.bn.Sample5_10.Constant.*;        //引入相关类
public class Cube {
    ColorRect cr;                                   //用于绘制各个面的颜色矩形
    public Cube(MySurfaceView mv) {
        cr = new ColorRect(mv);                     //创建用于绘制各个面的颜色矩形对象
    }
    public void drawSelf() {
```

```
    MatrixState.pushMatrix();                    //保护现场
        //绘制前小面
        MatrixState.pushMatrix();                //保护现场
        MatrixState.translate(0, 0, UNIT_SIZE);  //移到相应位置
        cr.drawSelf();                           //绘制矩形面
        MatrixState.popMatrix();                 //恢复现场
        ……//此处省略了绘制其他5个面的代码，套路与上述绘制前小面的基本一致
        MatrixState.popMatrix();                 //恢复现场
}}
```

提示

从上述代码中可以看出，本案例中Cube类的实现机制与前面各种变换一节案例中的Cube类不同。前面案例中的Cube类是通过给出立方体6个面中所有三角形的顶点来进行绘制的，而本案例中采用的是将一个颜色矩形平移、旋转到立方体6个面的位置，绘制6个颜色矩形来搭建的立方体。

（4）完成了Cube类的开发后，下面需要开发的就是颜色矩形类——ColorRect了。由于该类与3.1.2节中案例Sample3_1中的Triangle类非常相似，因此这里只介绍区别较大的drawSelf方法，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_10/com/bn/Sample5_10目录下的ColorRect.java。

```
public void drawSelf() {
    GLES20.glUseProgram(mProgram);                    //指定使用某套着色器程序
    GLES20.glUniformMatrix4fv(                        //将最终变换矩阵传入渲染管线
            muMVPMatrixHandle, 1, false, MatrixState.getFinalMatrix(), 0);
    GLES20.glUniformMatrix4fv(                        //将平移、旋转变换矩阵传入渲染管线
            muMMatrixHandle, 1, false, MatrixState.getMMatrix(), 0);
    GLES20.glVertexAttribPointer(                     //将顶点位置数据传入渲染管线
            maPositionHandle, 3, GLES20.GL_FLOAT, false, 3 * 4, mVertexBuffer);
    GLES20.glVertexAttribPointer(                     //将顶点着色数据传入渲染管线
                maColorHandle, 4, GLES20.GL_FLOAT, false, 4 * 4, mColorBuffer);

        GLES20.glEnableVertexAttribArray(maPositionHandle); //启用顶点位置数据
        GLES20.glEnableVertexAttribArray(maColorHandle);//启用顶点着色数据
        GLES20.glDrawArrays(GLES20.GL_TRIANGLE_FAN, 0, vCount);//绘制颜色矩形
  }
```

提示

从上述代码中可以看出，此方法与前面案例最大的区别是不但将最终变换矩阵传入了渲染管线，还将携带平移、旋转信息的变换矩阵传入了渲染管线。这样做的目的是为了在着色器中实现特效服务，详细情况会在后面介绍。

（5）细心的读者会发现，上一步骤中调用了MatrixState类中以前没有的方法getMMatrix，下面继续对MatrixState类进行升级，增加此方法，具体代码如下。

```
public static float[] getMMatrix()     //此方法功能为返回当前变换矩阵的内容
{
       return currMatrix;
```

```
}
```

（6）完成了本案例中Java代码的开发后，就可以开始开发着色器了。首先开发的是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_10/com/bn/Sample5_10_/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                          //总变换矩阵
uniform mat4 uMMatrix;                            //变换矩阵
attribute vec3 aPosition;                         //顶点位置
attribute vec4 aColor;                            //顶点颜色
varying vec4 vColor;                              //用于传递给片元着色器的颜色
varying vec3 vPosition;                           //用于传递给片元着色器的顶点位置
void main() {
  gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点位置
  vColor = aColor;                                //将接收的颜色传递给片元着色器
      vPosition=(uMMatrix * vec4(aPosition,1)).xyz;
                                                  //计算出此顶点变换后的位置传递给片元着色器
  }
```

- 此顶点着色器的大部分代码与前面案例中的相同，主要的区别就是增加了计算变换后的顶点位置，并将其通过易变变量传入片元着色器的功能。
- 变换后顶点的位置是通过从渲染管线传入的变换矩阵（uMMatrix）与顶点位置（aPosition）计算生成的。

提示

读者可能会产生这样的困扰："通过总变换矩阵与顶点位置不是已经计算出了变换后的顶点位置了吗？"。要注意的是这两种顶点位置是大有区别的：由于总变换矩阵中携带了摄像机、投影、普通变换的所有信息，因此通过总变换矩阵与顶点位置计算出来的是经过摄像机观察和投影以后已经位于近平面上的位置，可以理解为这时的顶点坐标已经是二维的了。而变换矩阵中仅携带了平移、旋转、缩放等普通变换的信

息，因此通过变换矩阵与顶点位置计算出来的是顶变换后在3D空间中的位置。

（7）完成了顶点着色器的开发后，就可以进行片元着色器的开发了，其代码如下。与前面的案例相似，此片元着色器的功能也是计算出片元的最终颜色。不同的是，前面的案例中是直接将顶点着色器传递过来的经管线插值后的颜色作为最终颜色，而这里还需要根据片元的位置加上淡红色。

代码位置：见随书光盘中源代码/第5章/Sample5_10/com/bn/Sample5_10_/assets目录下的frag.sh。

```
1 precision mediump float;
2 varying  vec4 vColor;                    //接收从顶点着色器过来的参数
3 varying vec3 vPosition;                  //接收从顶点着色器过来的顶点位置
4 void main() {
5   vec4 finalColor=vColor;
6   mat4 mm=mat4(                          //绕 z 轴转 20°的旋转变换矩阵
7           0.9396926,-0.34202012,0.0,0.0,  0.34202012,0.9396926,0.0,0.0,
8             0.0,0.0,1.0,0.0,  0.0,0.0,0.0,1.0);
9   vec4 tPosition=mm*vec4(vPosition,1);   //将顶点坐标绕 z 轴转 20°
10      if(mod(tPosition.x+100.0,0.4)>0.3) {        //计算 x 方向在不在红光色带范围内
11        finalColor=vec4(0.4,0.0,0.0,1.0)+finalColor;  //若在给最终颜色加上淡红色
12      }
13      gl_FragColor = finalColor;                    //给此片元颜色值
14   }
```

- 第6-9行首先声明了一个绕z 轴转20° 的旋转变换矩阵，然后将此矩阵与接收到的由渲染管线传入的片元位置进行计算，得到变换后的片元位置。
- 第10-12行根据此片元变换后空间位置的X 坐标是否位于一定的区间内来决定是否为最终颜色加上淡红色。

提示

之所以要绕Z轴旋转20°是为了得到斜着的红色光带的效果，参见图5-34。运行本案例后，读者若用手在屏幕上水平滑动，会发现两个立方体会随着手指的滑动绕*Y*轴旋转，但立方体上面的红色光带并不随着手指的滑动而改变位置，就像立方体位于远处固定位置条纹灯的照射下一样。

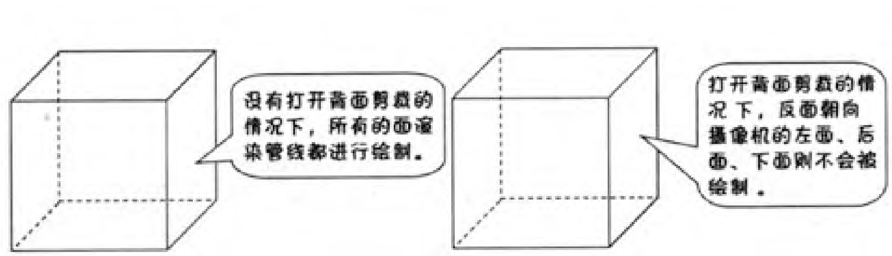


图5-34　是否打开背面剪裁的对比

通过本案例，读者一方面应该了解设置合理视角的重要性，另一方面也应该逐渐体会到OpenGLES2.0可编程渲染管线的强大之处，只要灵活运用就可以开发出非常多逼真的特效。了解OpenGL ES 1.x的读者会体会到，本案例中的条纹灯效果在OPen GL ES 1.x的平台上是不太容易实现的。而本案例中仅用了几行着色器代码就轻松地实现了，这也恰恰说明了可编程渲染管线的强大。

5.6　卷绕和背面剪裁

前面章节的有些例子中调用“GLES20.glEnable（GLES20.GL_CULL_FACE）；”语句打开了背面剪裁，但到目前为止并没有对背面剪裁进行介绍，本节将向读者介绍背面剪裁的相关知识。

5.6.1　基本知识

所谓背面剪裁是指渲染管线在对构成立体物体的三角形图元进行绘制时，仅当摄像机观察点位于三角形正面的情况下才绘制三角形，若观察点位于背面则不进行绘制。打开背面剪裁后在大部分情况下可以提高渲染效率，去除大量不必要的渲染工作，图5-34说明了这个问题。

从图5-34中可以看出，打开背面剪裁后正面朝向观察点的前面、上面、右面会被管线绘制，而反面朝向观察点的左面、后面、下面则不会被绘制。对于大部分封闭立体物体而言，这就实现了让管线对被挡住的面不执行绘制，有助于提高渲染速度。

若不打开背面剪裁，管线会对所有的面都进行绘制。被遮挡面上的片元虽然被绘制了，但还是会被遮挡面上的片元所覆盖，最终并不会出现在屏幕上。这导致很多绘制工作都白做了，宝贵的计算资源被浪费，同样情况下应用程序的帧速率（FPS）可能会下降很多。

> **提示**
>
> 对于封闭立体物体的渲染，一般情况下应该打开背面剪裁。但如果绘制的是平面物体，希望在正面和反面观察都能看到就不应该打开背面剪裁了。实际开发中，读者应该根据具体需要来决定。

了解了背面剪裁的功效后，很重要的一点就是如何确定摄像机是位于一个面的正面还是反面。在没有做出特殊设置的情况下，OpenGL ES中规定当摄像机观察一个三角形面时，若三角形中3个顶点的卷绕顺序是逆时针则摄像机观察其正面，反之摄像机观察其反面，图5-35说明了这个问题。

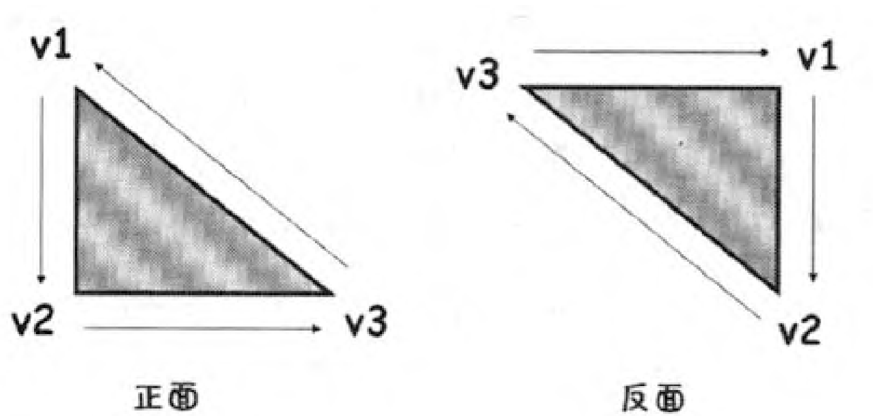


图5-35　通过卷绕顺序确定三角形的正反面

开发人员也可以通过设置告知渲染管线以顺时针卷绕为正面，以逆时针卷绕为反面。下面就将与背面剪裁及卷绕方式有关的语句一一列出，供读者在开发中根据具体情况选用。

- 打开背面剪裁的语句。

```
GLES20.glEnable(GLES20.GL_CULL_FACE);        // 打开背面剪裁
```

- 关闭背面剪裁的语句。

```
GLES20.glDisable(GLES20.GL_CULL_FACE);        //关闭背面剪裁
```

- 设置逆时针卷绕为正面的语句。

```
GLES20.glFrontFace(GLES20.GL_CCW);        //这是默认的，因此一般不用明确设置
```

- 设置顺时针卷绕为正面的语句。

```
GLES20.glFrontFace(GLES20.GL_CW);        //设置顺时针卷绕为正面，非默认，需要时采用
```

5.6.2　一个简单的案例

了解了背面剪裁与卷绕的基本原理后，下面本小节将通过一个简单的案例——Sample5_11来进一步进行介绍，其运行效果如图5-36所示。



图5-36　案例Sample5_11的运行效果图

从图5-36中可以看出，在关闭背面剪裁的情况下，无论采用哪种卷绕方式两个三角形都将被绘制。而打开背面剪裁后，随着卷绕方式的改变，只有在当前卷绕方式下正面面对摄像机的三角形才会被绘制。

> **提示**
>
> 此案例中左右两个三角形顶点的卷绕顺序是相反的，读者可以参考后面的代码。

了解了本案例的运行效果后，就可以进行代码的开发了。由于本案例中大部分代码在前面的案例中已经出现过，因此这里只介绍重要的且有明显区别的部分，具体如下所列。

（1）首先介绍TrianglePair（三角形对，代表两个三角形）类中负责初始化顶点坐标与着色数据的initVertexData方法，其代码如下。

代码位置：见随书光盘中源代码/第5章/Sample5_11/com/bn/Sample5_11目录下的TrianglePair.java。

```
1 public void initVertexData() {                       //初始化顶点坐标与着色数据的方法
2     vCount = 6;                                        //顶点数量
3     float vertices[] = new float[] {
4             -8 * UNIT_SIZE, 10 * UNIT_SIZE, 0,         //左面三角形的第1个顶点
5             -2 * UNIT_SIZE, 2 * UNIT_SIZE, 0,          //左面三角形的第2个顶点
6             -8 * UNIT_SIZE, 2 * UNIT_SIZE, 0,          //左面三角形的第3个顶点
7             8 * UNIT_SIZE, 2 * UNIT_SIZE, 0,           //右面三角形的第1个顶点
8             8 * UNIT_SIZE, 10 * UNIT_SIZE, 0,          //右面三角形的第2个顶点
9             2 * UNIT_SIZE, 10 * UNIT_SIZE, 0           //右面三角形的第3个顶点
10        };
11        ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);
                                                         //创建顶点坐标数据缓冲
12        vbb.order(ByteOrder.nativeOrder());            //设置字节顺序
13        mVertexBuffer = vbb.asFloatBuffer();           //转换为Float型缓冲
14        mVertexBuffer.put(vertices);                   //向缓冲区中放入顶点坐标数据
15        mVertexBuffer.position(0);                     //设置缓冲区起始位置
16        ……//此处省略了初始化顶点颜色数据缓冲的代码，读者可自行查看随书光盘中的源代码
17 }
```

说明

上述代码中第4-9行给出了两个三角形各个顶点的坐标，读者若有兴趣拿出坐标纸简单勾勒一下便会发现，两个三角形顶点的卷绕顺序是相反的。

（2）接着介绍MySurfaceView的内部类SceneRenderer中负责渲染每一帧画面的onDrawFrame方法，其代码如下。

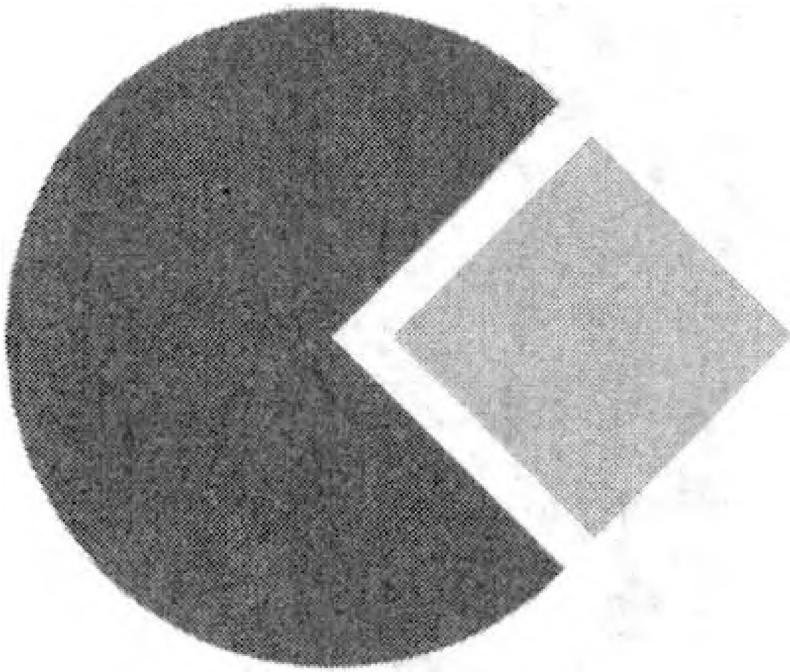
代码位置：见随书光盘中源代码/第5章/Sample5_11/com/bn/Sample5_11目录下的MySurfaceView.java。

```
1 public void onDrawFrame(GL10 gl) {
2     if (cullFaceFlag) {                              //判断是否要打开背面剪裁
3         GLES20.glEnable(GLES20.GL_CULL_FACE);        //打开背面剪裁
4     } else {
5         GLES20.glDisable(GLES20.GL_CULL_FACE);       //关闭背面剪裁
6     }
7     if (cwCcwFlag) {                                 //判断是否需要打开顺时针卷绕
8         GLES20.glFrontFace(GLES20.GL_CCW);           //使用逆时针卷绕
9     } else {
10            GLES20.glFrontFace(GLES20.GL_CW);        //使用顺时针卷绕
11        }
12        GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT    //清除深度缓冲与颜色缓冲
13                | GLES20.GL_COLOR_BUFFER_BIT);
14        MatrixState.pushMatrix();                    //保护现场
15        MatrixState.translate(0, -1.4f, 0);          //沿y轴负方向平移
16        tp.drawSelf();                               //绘制三角形对
17        MatrixState.popMatrix();                     //恢复现场
18    }
```

- 第2-6行中通过一个布尔型变量cullFaceFlag来控制打开或关闭背面剪裁，此变量通过用户单击界面中的单选按钮来改变。
- 第7-11行中通过一个布尔型变量cwCcwFlag来设置卷绕方式，此变量也通过用户单击界面中的单选按钮来改变。

5.7　本章小结

本章向读者介绍了很多基于OpenGL ES 2.0平台进行3D应用开发必知必会的基本知识，主要包括：两种投影方式、3种基本变换、7种绘制方式以及设置合理的视角与背面剪裁。掌握了这些知识后，读者进行3D应用程序开发的能力应该得到了进一步的提升，为继续学习后面章节中更高级的内容打下坚实的基础。



第6章　光照

通过前面章节的学习，读者已经有能力基于OpenGL ES 2.0开发出简单的3D场景了。但对于场景中的物体只能通过直接给出颜色的方式进行着色渲染，真实感较差。本章将向读者介绍光照效果的开发，通过本章的学习，读者可以为场景中的物体增加逼真的光照效果，大大提升了场景的真实感。

6.1　曲面物体的构建

前面的章节中已经介绍了如何构建3D物体，但已给出案例中的3D物体基本都是平面性质的（直接的平面形状或长方体等），还没有曲

面性质的物体。对于演示光照效果而言，曲面物体更能凸显出光照效果的强大。因此在正式介绍光照之前，本节首先基于球体的构建向读者简单介绍一下曲面物体的构建策略。

6.1.1　球体构建的基本原理

通过上一章的学习读者已经知道，OpenGL ES中任何形状的3D物体都是用三角形拼凑而成的，因此构建曲面物体最重要的就是找到将曲面恰当拆分成三角形的策略。最基本的策略是首先按照一定的规则将物体按行和列两个方向进行拆分，这时就可以得到很多的小四边形。然后再将每个小四边形拆分成两个三角形即可，图6-1给出了基于这种策略的球面拆分思路。

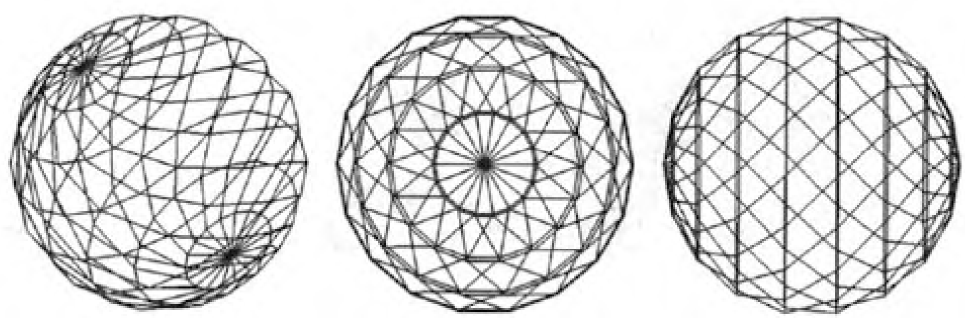


图6-1　球的表面也是由平面组成的

从图6-1中可以看出，球面首先被按照纬度（行）和经度（列）的方向拆分成了很多的小四边形，每个小四边形又被拆分成两个小三角形。这种拆分方式下，三角形中每个顶点的坐标都可以用解析几何的公式方便地计算出来，具体情况如下。

$$x = R\times\cos\alpha\times\cos\beta;\qquad y = R\times\cos\alpha\times\sin\beta;\qquad z = R\times\sin\alpha$$

上述给出的是当球的半径为R，在纬度为α，经度为β处球面上顶点坐标的计算公式。

说明

对一个曲面物体进行拆分时，以何为行，以何为列是不一定的，读者应该根据具体情况做出选择。

对曲面物体进行拆分时，拆分得越细，最终的绘制结果就越接近真实情况，图6-2很好地说明了这个问题。

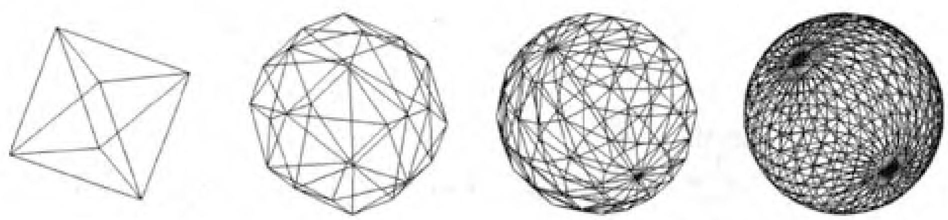


图6-2　切分得越细就越接近于球面

提示

图6-2中从左至右依次为按照90°/份、45°/份、22.5°/份、11.25°/份对球面进行切分的情况，可以明显地看出，切分得越细，就越接近于真实的曲面。但也不是越细越好，切分得太细就会造成顶点数量过多，渲染速度大大降低。因此在开发中读者要掌握好两者之间的平衡，兼顾速度与效果。

6.2.2　案例效果概览

上一小节已经介绍了如何将球面拆分成一组小三角形，下面就可以基于上一小节介绍的原理开发出球体绘制的案例Sample6_1了。正式开发代码之前，有必要首先了解一下本案例的运行效果，如图6-3所示。

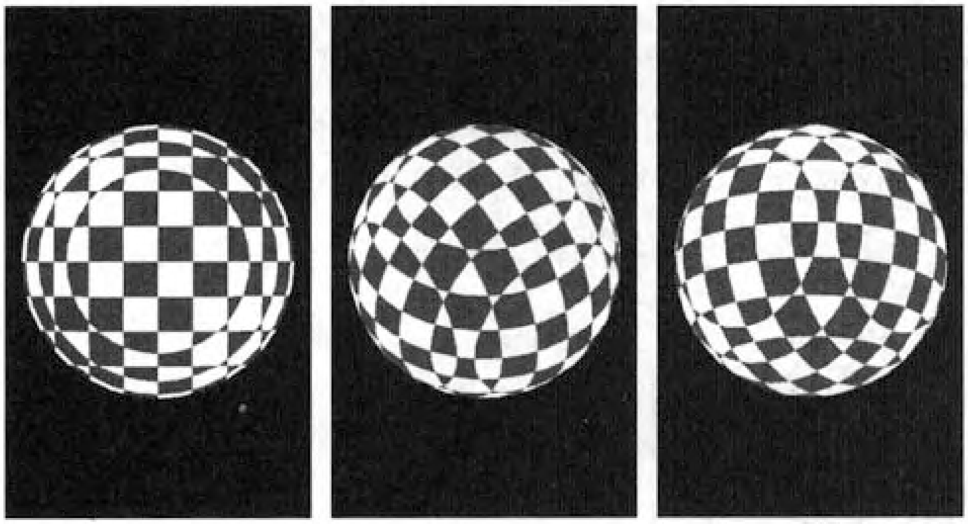


图6-3　案例Sample6_1运行效果

从图6-3中可以看出，本案例中的球体不是用单一颜色进行着色的，采用的是棋盘纹理着色器。棋盘纹理着色器是一种非常简单的着色器，其原理如图6-4所示。

说明

图6-4中的立方体为球的外接立方体，球面上的每个位置都在此外接立方体之内，此外接立方体沿X、Y、Z轴方向被切分成了很多同样尺寸的小方块。

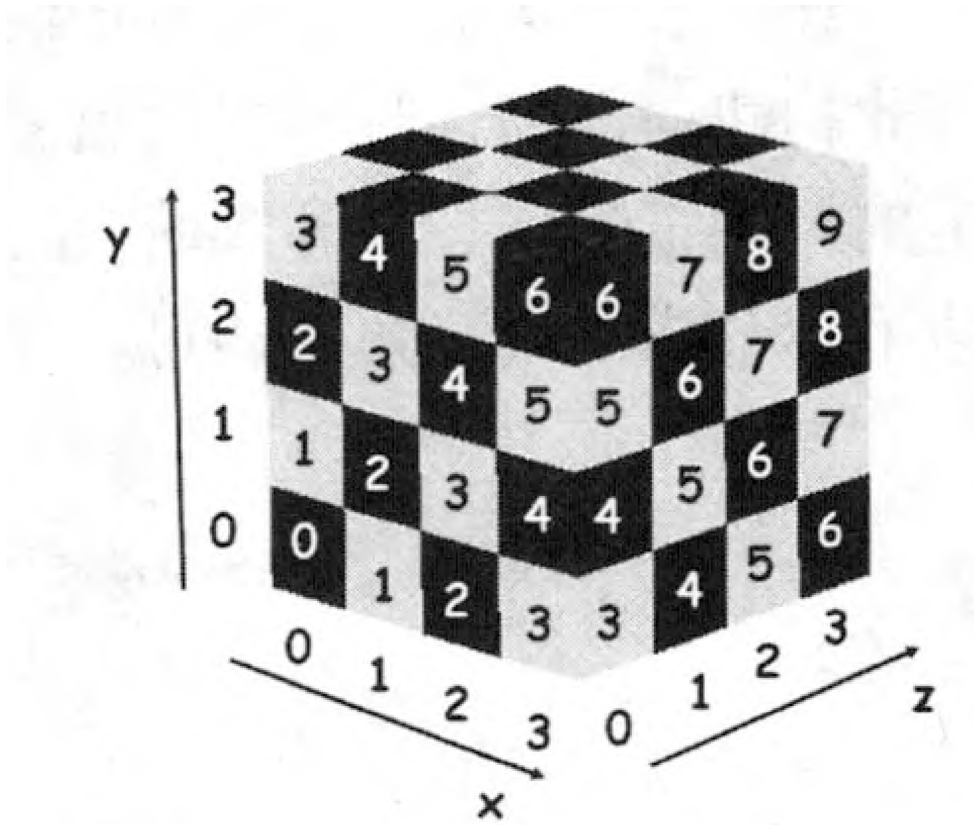


图6-4　棋盘纹理着色器的原理

具体的着色策略为，若片元位于黑色小方块中，就将该片元的颜色设置为红色；若片元位于浅灰色小方块中则将片元的颜色设置为白色，具体计算方法如下所列。

- 首先计算出当前片元X、Y、Z坐标对应的行数（*X* 轴）、层数（*Y* 轴）及列数（*Z* 轴）。
- 如果行数、层数、列数之和为奇数，则片元为红色；若和为偶数，则片元为白色。

6.2.3　开发步骤

了解了案例的运行效果与基本原理后，就可以进行代码的开发了，具体步骤如下。

提示

由于本案例中的很多类与前面章节案例中的很类似，因此在这里只给出本案例中具有特殊性及代表性的代码。若读者对省略的代码感兴趣，可以参考随书光盘。

（1）首先需要介绍的是负责按照切分规则生成球面上顶点的坐标，并渲染球体的Ball类，其代码框架如下。

代码位置：见随书光盘中源代码/第6章/Sample6_1/com/bn/Sample6_1目录下的Ball.java。

```java
package com.bn.Sample6_1;                                   //声明包
import java.nio.ByteBuffer;                                 //引入相关类
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
import android.opengl.GLES20;                               //引入相关类
public class Ball {                                         //球
   int mProgram;                                            //自定义渲染管线着色器程序 id
   int muMVPMatrixHandle;                                   //总变换矩阵引用
   int maPositionHandle;                                    //顶点位置属性引用
   int muRHandle;                                           //球的半径参数引用
      String mVertexShader;                                 //顶点着色器代码脚本
      String mFragmentShader;                               //片元着色器代码脚本
      FloatBuffer mVertexBuffer;                            //顶点坐标数据缓冲
      int vCount = 0;                                       //顶点数量
      float yAngle = 0;float xAngle = 0;float zAngle = 0;  //绕 X、Y、Z 轴旋转的角度
      float r = 0.8f;                                       //球的半径
      public Ball(MySurfaceView mv) {
         initVertexData();                                  //初始化顶点坐标
         initShader(mv);                                    //初始化着色器
      }
      public void initVertexData() {
         ……//此处省略了初始化顶点坐标方法的代码，将在后面步骤中介绍
      }
      public void initShader(MySurfaceView mv) {           //初始化着色器的方法
         ……//此处省略了部分代码，读者可自行查看随书光盘的源代码
         //获取着色器程序中球半径参数的引用
         muRHandle = GLES20.glGetUniformLocation(mProgram, "uR");
      }
      public void drawSelf() {
         ……//此处省略了部分代码，读者可自行查看随书光盘的源代码
         GLES20.glUniform1f(muRHandle, r * UNIT_SIZE);   //将半径属性传入渲染管线
```

```
31          GLES20.glVertexAttribPointer(              //将顶点属性传入渲染管线
32              maPositionHandle, 3, GLES20.GL_FLOAT, false, 3 * 4, mVertexBuffer);
33          GLES20.glEnableVertexAttribArray(maPositionHandle);//允许渲染管线顶点位置数据
34          GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount);//绘制球
35  }}
```

- 第6-15行声明了一些成员变量，其中大部分都与前面的很多案例类似。最大的区别是增添了着色器中球半径参数的引用muRHandle以及球的半径r。
- 第23-27行为初始化着色器的initShader方法，其中大部分代码与前面很多的案例类似，主要区别为增加了获取着色器程序中球半径参数引用的代码。
- 第28-35行为绘制球体的drawSelf方法，与前面案例中此方法的主要区别是增加了将当前球的半径传入渲染管线的代码。

（2）接下来介绍上一步骤中省略的负责初始化顶点的initVertexData方法，其具体代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_1/com/bn/Sample6_1目录下的Ball.java。

```
public void initVertexData() {
  ArrayList<Float> alVertix = new ArrayList<Float>();//存放顶点坐标值的 ArrayList
  final int angleSpan = 10;                          //将球进行单位切分的角度
  for (int vAngle = -90; vAngle < 90; vAngle = vAngle + angleSpan) {
                                                     //纬度方向 angleSpan 度一份
    for (int hAngle = 0; hAngle <= 360; hAngle = hAngle + angleSpan){
                                                     //经度方向 angleSpan 度一份
          //计算出以当前经度、纬度位置的顶点为左上侧点的四边形 4 个顶点的坐标
          float x0 = (float) (r * UNIT_SIZE                  //第 1 个顶点的坐标
                  * Math.cos(Math.toRadians(vAngle)) * Math.cos(Math
                  .toRadians(hAngle)));
              float y0 = (float) (r * UNIT_SIZE
                      * Math.cos(Math.toRadians(vAngle)) * Math.sin(Math .
                      toRadians(hAngle)));
              float z0 = (float) (r * UNIT_SIZE * Math.sin(Math .toRadians(vAngle)));
              float x1 = (float) (r * UNIT_SIZE           //第 2 个顶点的坐标
                      * Math.cos(Math.toRadians(vAngle)) * Math.cos(Math
                      .toRadians(hAngle + angleSpan)));
              float y1 = (float) (r * UNIT_SIZE
                      * Math.cos(Math.toRadians(vAngle)) * Math.sin(Math
                      .toRadians(hAngle + angleSpan)));
              float z1 = (float) (r * UNIT_SIZE * Math.sin(Math .toRadians(vAngle)));
              float x2 = (float) (r * UNIT_SIZE           //第 3 个顶点的坐标
                      * Math.cos(Math.toRadians(vAngle + angleSpan)) * Math
                      .cos(Math.toRadians(hAngle + angleSpan)));
              float y2 = (float) (r * UNIT_SIZE
                      * Math.cos(Math.toRadians(vAngle + angleSpan)) * Math
                      .sin(Math.toRadians(hAngle + angleSpan)));
              float z2 = (float) (r * UNIT_SIZE * Math.sin(Math .toRadians(vAngle
              + angleSpan)));
              float x3 = (float) (r * UNIT_SIZE           //第 4 个顶点的坐标
                      * Math.cos(Math.toRadians(vAngle + angleSpan)) * Math
                      .cos(Math.toRadians(hAngle)));
              float y3 = (float) (r * UNIT_SIZE
```

```
31                      * Math.cos(Math.toRadians(vAngle + angleSpan)) * Math
32                      .sin(Math.toRadians(hAngle)));
33              float z3 = (float) (r * UNIT_SIZE * Math.sin(Math .toRadians(vAngle
                + angleSpan)));
34              //将4个顶点的坐标按照卷绕成两个三角形的需要依次存入列表
35              alVertix.add(x1); alVertix.add(y1); alVertix.add(z1);
36              alVertix.add(x3); alVertix.add(y3); alVertix.add(z3);
37              alVertix.add(x0); alVertix.add(y0); alVertix.add(z0);
38              alVertix.add(x1); alVertix.add(y1); alVertix.add(z1);
39              alVertix.add(x2); alVertix.add(y2); alVertix.add(z2);
40              alVertix.add(x3); alVertix.add(y3); alVertix.add(z3);
41          }}
42          vCount = alVertix.size() / 3; //顶点的数量为坐标值数量的1/3，因为一个顶点有3个坐标
43          float vertices[] = new float[vCount * 3];
44          for (int i = 0; i < alVertix.size(); i++) {
                                         //将alVertix列表中的坐标值转存到一个float型数组中
45              vertices[i] = alVertix.get(i);
46          }
47          ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);
                                                              //创建顶点坐标数据缓冲
48          vbb.order(ByteOrder.nativeOrder());               //设置字节顺序
49          mVertexBuffer = vbb.asFloatBuffer();              //转换为int型缓冲
50          mVertexBuffer.put(vertices);                      //向缓冲区中放入顶点坐标数据
51          mVertexBuffer.position(0);                        //设置缓冲区起始位置
52      }
```

- 第3行中的angleSpan为将球进行经纬度方向单位切分的角度，角度越小，切分得就越细，绘制出来的形状也越接近于球。
- 第4-41行用双层for循环将球按照一定的角度跨度（angleSpan）沿纬度、经度方向进行切分。每次循环到一组纬度、经

度时都将对应顶点看作小四边形的左上侧点，然后按照规律计算出小四边形中其他3个顶点的坐标，最后按照需要将用于卷绕成两个三角形的6个顶点的坐标依次存入列表。

- 第42-51行首先将顶点坐标数据转存进数组中，然后再存入顶点数据缓冲中。

（3）完成了Java代码的开发后，就可以进行着色器的开发了。首先是顶点着色器，其具体代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_1/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                                 //总变换矩阵
attribute vec3 aPosition;                                //从管线接收的顶点位置
varying vec3 vPosition;                                  //用于传递给片元着色器的顶点位置
void main() {
  gl_Position = uMVPMatrix * vec4(aPosition,1);  //根据总变换矩阵计算此次绘制此顶点的位置
  vPosition = aPosition;                                 //将原始顶点位置传递给片元着色器
}
```

> **提示**
>
> 上述顶点着色器的代码与前面章节的案例基本一致，主要是增加了将顶点位置通过易变变量vPosition传递给片元着色器的相关代码。

（4）完成了顶点着色器的开发后，就可以开发本案例中实现棋盘着色的片元着色器了，其代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_1/assets目录下的frag.sh。

```
precision mediump float;                          //指定浮点相关变量的精度
uniform float uR;                                 //从宿主程序中传入的球半径
varying vec3 vPosition;                           //接收从顶点着色器传递过来的顶点位置
void main() {
  vec3 color;
  float n = 8.0;                                  //外接立方体每个坐标轴方向切分的份数
  float span = 2.0*uR/n;                          //每一份的尺寸(小方块的边长)
  int i = int((vPosition.x + uR)/span);           //当前片元位置小方块的行数
  int j = int((vPosition.y + uR)/span);           //当前片元位置小方块的层数
      int k = int((vPosition.z + uR)/span);       //当前片元位置小方块的列数
      int whichColor = int(mod(float(i+j+k),2.0));
                                    //计算当前片元行数、层数、列数的和并对2取模
      if(whichColor == 1) {                       //奇数时为红色
             color = vec3(0.678,0.231,0.129);     //红色
      }else {                                     //偶数时为白色
             color = vec3(1.0,1.0,1.0);           //白色
      }
      gl_FragColor=vec4(color,0);                 //将计算出的片元颜色传递给管线
  }
```

说明

上述片元着色器实现了如前一小节图6-4所示的棋盘着色器,其根据片元的位置计算出片元所在小方块的行数、层数、列数，再根据3个数之和的奇偶性确定片元所采用的颜色。

6.2　基本光照效果

完成了上一节中球体的开发后，下面的部分将基于此球体逐步介绍光照各个方面的知识。具体计划为首先介绍OpenGL ES 2.0中光照模型的基本知识，然后逐步为球体添加不同的光照效果。

6.2.1 光照的基本模型

如果要用一个数学模型完全真实地描述现实世界中的光照是很难的，一方面数学模型本身可能太过复杂，另一方面复杂的模型可能导致巨大的计算量。因此OpenGL ES 2.0中采用的光照模型相对现实世界进行了很大的简化，将光照分成了3种组成元素（也可以称为3个通道），包括环境光、散射光以及镜面光，具体情况如图6-5所示。

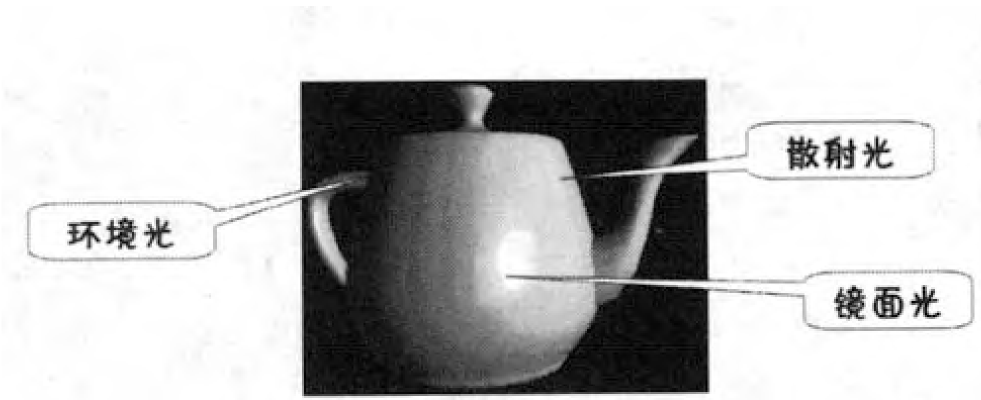


图6-5 光的3个通道

> **提示**
>
> 实际开发中，3个光照通道是分别采用不同的数学模型独立计算的，下面的几个小节将一一进行详细介绍。

6.2.2 环境光

环境光（Ambient）指的是从四面八方照射到物体上，全方位360°都均匀的光。其代表的是现实世界中从光源射出，经过多次反射后，各方向基本均匀的光。环境光最大的特点是不依赖于光源的位置，而且没有方向性，图6-6简单地说明了这个问题。

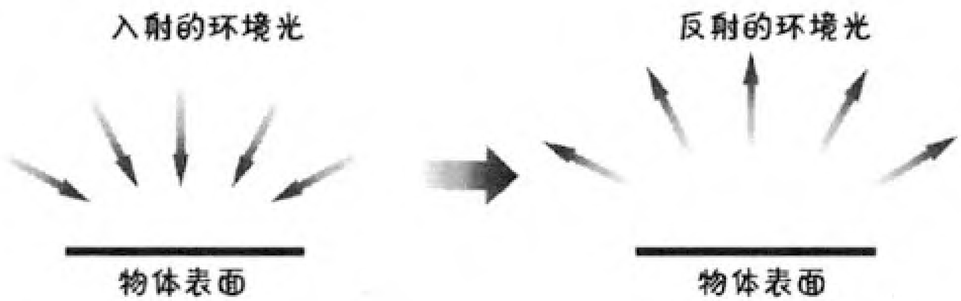
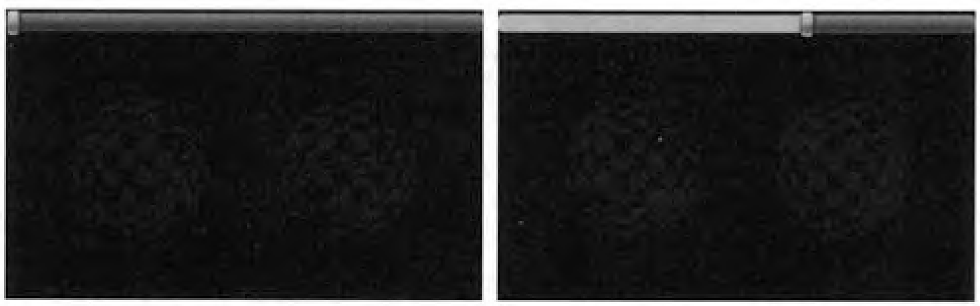


图6-6　环境光的基本情况

从图6-6中可以看出，环境光不但入射是均匀的，反射也是各向均匀的。用于计算环境光的数学模型非常简单，具体公式如下。

环境光照射结果=材质的反射系数×环境光强度

了解了环境光的基本原理后，下面将通过一个简单的案例Sample6_2来介绍环境光效果的开发，案例的运行效果如图6-7所示。


图6-7　案例Sample6_2的运行效果图

说明

图6-7界面上部的滑块用来控制光源的位置，由于环境光的照射效果与光源位置无关，故无论光源调节到什么位置效果都相同。另外，实际开发中环境光强度一般都设置得较弱，因此仅用环境光照射的物体看起来并不是很清楚。

了解了案例的运行效果后，就可以进行代码的开发了。由于本案例主要是对上一节中Sample6_1的升级，因此这里仅给出变化较大且有代表性的部分，具体如下所列。

（1）首先需要修改的是MySurfaceView类中的onDrawFrame方法，具体代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_2/com/bn/Sample6_2目录下的MySurfaceView.java。

```
public void onDrawFrame(GL10 gl) {
    GLES20.glClear(                                      //清除深度缓冲与颜色缓冲
        GLES20.GL_DEPTH_BUFFER_BIT | GLES20.GL_COLOR_BUFFER_BIT);
    MatrixState.pushMatrix();                            //保护现场
    MatrixState.pushMatrix();

    MatrixState.translate(-1.2f, 0, 0);                  //沿x轴负方向平移
    ball.drawSelf();                                     //绘制球
    MatrixState.popMatrix();
    MatrixState.pushMatrix();
        MatrixState.translate(1.2f, 0, 0);               //沿x轴正方向平移
        ball.drawSelf();                                 //绘制球
        MatrixState.popMatrix();
        MatrixState.popMatrix();                         //恢复现场
  }
```

提示

从上述代码中可以看出，此方法主要的变化是将原来只绘制一次球体更改为在左右两侧各绘制一次球体。绘制两次球体的目的是为后面开发其他光照效果服务，这里读者不必深究。

（2）在MySurfaceView类中添加了一个成员变量lightOffset，用来表示光源的X 轴坐标。同时在Activity中为滑块添加了监听器，其功能为当滑块被拖动时修改lightOffset的值。

> **提示**
>
> 由于环境光与光源位置无关，故在本案例中拖拉滑块时场景没有变化，但后面开发其他光照效果时就很有作用了。

（3）完成了Java代码的修改后，接着就可以开发着色器了。首先是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_2/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                              //总变换矩阵
attribute vec3 aPosition;                             //顶点位置
varying vec3 vPosition;                               //用于传递给片元着色器的顶点位置
varying vec4 vAmbient;                                //用于传递给片元着色器的环境光分量
void main() {
   gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点的位置
   vPosition = aPosition;                             //将顶点的位置传递给片元着色器
   vAmbient = vec4(0.15,0.15,0.15,1.0);               //将环境光强度传递给片元着色器
}
```

> **提示**
>
> 上述代码中主要是增加了将环境光强度传递给片元着色器的代码。由于本案例比较简单，环境光的强度就固化在顶点着色器中

了，未来有需要可以改为由Java程序传给顶点或片元着色器中的相关一致变量。

（4）开发完顶点着色器后，接下来就可以开发片元着色器了，其具体代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_2/assets目录下的frag.sh。

```
precision mediump float;                      //给出浮点精度
uniform float uR;                             //球半径
varying vec3 vPosition;                       //接收从顶点着色器过来的顶点位置
varying vec4 vAmbient;                        //接收从顶点着色器过来的环境光强度
void main(){
  vec3 color;
  ……//此处省略了按照棋盘着色器规则计算片元颜色值的代码，与前面Sample6_1
  //案例中的完全相同，需要的读者请自行查看随书光盘中的源代码
  vec4 finalColor=vec4(color,0);              //最终颜色
      gl_FragColor=finalColor*vAmbient;       //根据环境光强度计算最终片元颜色值
  }
```

说明

上述片元着色器代码与前面案例的基本相同，主要是增加了接收环境光强度以及使用环境光强度与片元本身颜色值加权计算产生最终片元颜色值的相关代码。

6.2.3　散射光

上一小节中给出了仅仅使用环境光进行照射的案例，读者可能觉得效果并不好。确实如此，仅仅有环境光的场景效果是很差的，没有层次感。本节将介绍另外一种真实感好很多的光照效果——散射光（Diffuse），其指的是从物体表面向全方位360°均匀反射的光，如图6-8所示。

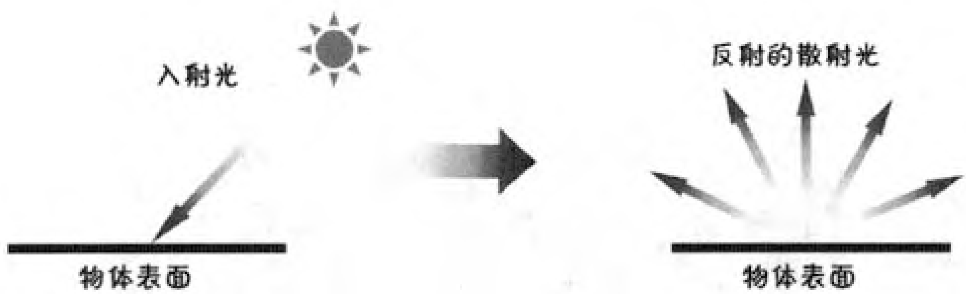


图6-8　散射光基本情况

散射光具体代表的是现实世界中粗糙的物体表面被光照射时，反射光在各个方向基本均匀（也称之为“漫反射”）的情况，图6-9很好地说明了这个问题。



图6-9　光在粗糙的表面上发生漫反射

虽然反射后的散射光在各个方向是均匀的，但散射光反射的强度与入射光的强度以及入射的角度密切相关。因此当光源的位置发生变

化时，散射光的效果会发生明显变化。主要体现为当光垂直地照射到物体表面时比斜照时要亮，其具体计算公式如下。

散射光照射结果=材质的反射系数×散射光强度×max（cos（入射角），0）

实际开发中往往分两步进行计算，此时公式被拆解为如下情况。

散射光最终强度=散射光强度×max（cos（入射角），0）

散射光照射结果=材质的反射系数×散射光最终强度

> **提示**
>
> 材质的反射系数实际指的就是物体被照射处的颜色，散射光强度指的是散射光中RGB（红、绿、蓝）3个色彩通道的强度。

从上述公式中可以看出，与环境光计算公式唯一的区别是引入了最后一项“max（cos（入射角），0）”。其含义是入射角越大，反射强度越弱，当入射角的余弦值为负时（即入射角大于90°），反射强度为0。由于入射角为入射光向量与法向量的夹角，因此其余弦值并不需要调用三角函数进行计算，只需要首先将两个向量进行规格化，然后再进行点积即可，图6-10说明了这个问题。

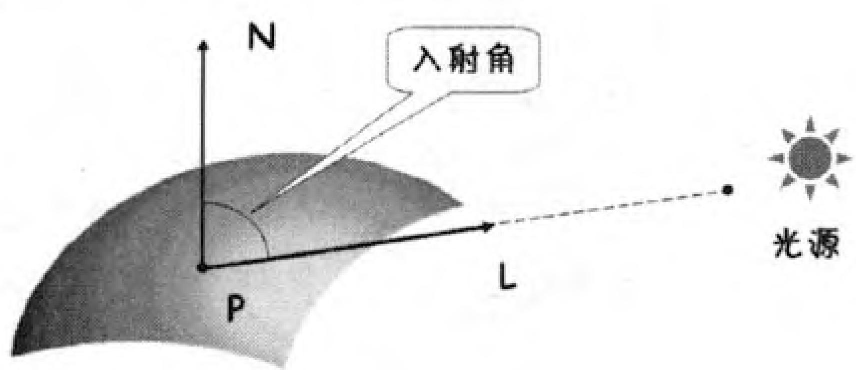


图6-10　散射光的计算

图6-10中的N代表被照射点表面的法向量，P为被照射点，L为从P点到光源的向量。N与L的夹角即为入射角。向量数学中，两个向量的点积为两个向量夹角的余弦值乘以两个向量的模，而规格化后向量的

模为1。因此首先将两个向量规格化，再点积就可以求得两个向量夹角的余弦值。

提示

由于本书篇幅有限，关于向量数学的相关问题不作详细讨论，有兴趣的读者可以参考其他相关数学资料或书籍。

了解了散射光的基本原理后，下面给出一个使用了散射光的案例Sample6_3，其运行效果如图6-11所示。



图6-11　案例Sample6_3的运行效果图

说明

图6-11左侧的图表示光源位于场景左侧进行照射的情况，右侧的图表示光源位于右侧进行照射的情况。从左右两幅效果图的对比中可以看出，正对光源（入射角小）的位置看起来较亮，而随着入射角的增大越来越暗，直至入射角大于90°后完全不能照亮。

了解了散射光的基本原理及案例的运行效果后，就可以进行案例的开发了。由于实际上本案例仅仅是将案例Sample6_2复制了一份并进行了修改，因此这里仅给出修改的主要步骤，具体如下所列。

（1）由于散射光效果与光源的位置密切相关，因此需要将光源的位置传递进着色器以进行光照的计算。为了方便起见，首先需要对工具类MatrixState进行升级，增加存储当前光源位置的相关成员变量以及设置光源位置的方法，具体代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_3/com/bn/Sample6_3目录下的MatrixState.java。

```
1 public static float[] lightLocation = new float[] { 0, 0, 0 };   //光源位置数组
2 public static FloatBuffer lightPositionFB;                        //光源位置的缓冲
```

```
3 static ByteBuffer llbbL = ByteBuffer.allocateDirect(3 * 4);       //待用的字节缓冲
4 public static void setLightLocation(float x, float y, float z) {  //设置光源位置的方法
5     llbbL.clear();                                                 //清除缓冲中原有的数据
6     lightLocation[0] = x;lightLocation[1] = y;lightLocation[2] = z;
                                                                     //将新的光源位置存入数组
7     llbbL.order(ByteOrder.nativeOrder());                          //设置字节顺序
8     lightPositionFB = llbbL.asFloatBuffer();                       //转换为 float 型缓冲
9     lightPositionFB.put(lightLocation);                            //将光源位置放入缓冲
10        lightPositionFB.position(0);                               //设置缓冲区起始位置
11    }
```

- 第1-3行为该类中新增加的成员变量，主要有光源位置数组、光源位置缓冲和待用字节缓冲。
- 第4-11行为设置光源位置的方法，该方法主要功能为将参数中传递过来的光源位置首先存入数组，然后再存放进对应的float型缓冲供渲染时传入管线。

（2）接着需要修改的是Ball类，主要是增加初始化法向量数据以及将法向量数据传入渲染管线的相关代码，具体情况如下。

代码位置：见随书光盘中源代码/第6章/Sample6_3/com/bn/Sample6_3目录下的Ball.java。

```java
public class Ball {
    ……//此处省略了部分代码，请读者自行查看随书光盘中的源代码
    int maNormalHandle;                                   //顶点法向量属性引用
    int maLightLocationHandle;                            //光源位置属性引用
    FloatBuffer mNormalBuffer;                            //顶点法向量数据缓冲引用
    public Ball(MySurfaceView mv) {/*代码省略*/}
    public void initVertexData() {                        //初始化顶点数据的方法
        ……//此处省略了部分代码，请读者自行查看随书光盘中的源代码
      ByteBuffer nbb = ByteBuffer.allocateDirect(vertices.length*4);//创建顶点法向量缓冲
         nbb.order(ByteOrder.nativeOrder());              //设置字节顺序
         mNormalBuffer = nbb.asFloatBuffer();             //转换为 float 型缓冲
         mNormalBuffer.put(vertices);                     //向缓冲区中放入顶点法向量数据
         mNormalBuffer.position(0);                       //设置缓冲区起始位置
        }
        public void initShader(MySurfaceView mv) {        //初始化着色器
            ……//此处省略了部分代码，请读者自行查看随书光盘中的源代码
          maNormalHandle= GLES20.glGetAttribLocation(   //获取顶点法向量属性变量引用
                mProgram, "aNormal");
          maLightLocationHandle=GLES20.glGetUniformLocation(
                mProgram, "uLightLocation");              //获取光源位置一致变量引用
        }
        public void drawSelf() {
            ……//此处省略了部分代码，请读者自行查看随书光盘中的源代码
            GLES20.glUniform3fv(                          //将光源位置传入渲染管线
                maLightLocationHandle, 1, MatrixState.lightPositionFB);
            GLES20.glVertexAttribPointer(                 //将顶点位置数据传入渲染管线
                    maPositionHandle, 3, GLES20.GL_FLOAT,false, 3 * 4, mVertexBuffer);
            GLES20.glVertexAttribPointer(                 //将顶点法向量数据传入渲染管线
                    maNormalHandle, 3, GLES20.GL_FLOAT, false,3 * 4, mNormalBuffer);
```

```
30            GLES20.glEnableVertexAttribArray(maPositionHandle); //启用顶点位置数据
31            GLES20.glEnableVertexAttribArray(maNormalHandle);   //启用顶点法向量数据
32            GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount);//绘制球体
33    }}
```

- 第3-5行增加了声明顶点法向量属性变量引用、光源位置一致变量引用及顶点法向量数据缓冲的代码。
- 第7-14行为初始化顶点数据的方法，其中增加了初始化顶点法向量数据缓冲的相关代码。由于本案例中原始情况下的球心位于坐标原点，所以每个顶点法向量的X、Y、Z轴分量与顶点的X、Y、Z坐标是一致的。这样就不必单独计算每个顶点的法向量了，直接将顶点坐标序列看作顶点法向量序列使用即可。
- 第15-21行在初始化着色器的方法中增加了获取顶点法向量属性变量引用以及光源位置一致变量引用的代码。
- 第22-33行为绘制球的drawSelf方法，其中增加了将法向量数据与光源位置数据传送进渲染管线的代码，同时也增加了启用顶点法向量数据的代码。

> **提示**
>
> 并不是所有的情况下顶点法向量与顶点坐标都有必然联系，很多情况下顶点的法向量需要单独给出，本书中后面会有很多这样的案例。

（3）接下来需要修改的是MySurfaceView类中的onDrawFrame方法，具体代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_3/com/bn/Sample6_3目录下的MySurfaceView.java。

```
public void onDrawFrame(GL10 gl) {
    GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT                //清除深度缓冲与颜色缓冲
            | GLES20.GL_COLOR_BUFFER_BIT);
    MatrixState.setLightLocation(lightOffset, 0, 1.5f);      //设置光源位置
    MatrixState.pushMatrix();                                //保护现场
    ……//此处省略了绘制两次球的代码，请读者自行查看随书光盘中的源代码
    MatrixState.popMatrix();                                 //恢复现场
}
```

提示

此方法变化不大，仅仅是在绘制每帧画面之前增加了设置光源位置的代码。这样当用户拖拉滑块改变光源的*X*坐标（lightOffset）时，绘制的画面就会随光源位置的变化而变化了。

（4）完成了MySurfaceView类的修改后，对Java代码的修改就基本完成了，下面需要修改的是着色器的代码。首先是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_3/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                  //总变换矩阵
uniform mat4 uMMatrix;                    //变换矩阵(包括平移、旋转、缩放)
uniform vec3 uLightLocation;              //光源位置
```

```
attribute vec3 aPosition;                                    //顶点位置
attribute vec3 aNormal;                                      //顶点法向量
varying vec3 vPosition;                                      //用于传递给片元着色器的顶点位置
varying vec4 vDiffuse;                                       //用于传递给片元着色器的散射光分量
void pointLight (                                            //散射光光照计算的方法
  in vec3 normal,                                            //法向量
     inout vec4 diffuse,                                     //散射光计算结果
     in vec3 lightLocation,                                  //光源位置
     in vec4 lightDiffuse                                    //散射光强度
   ){
     vec3 normalTarget=aPosition+normal;                     //计算变换后的法向量
     vec3 newNormal=(uMMatrix*vec4(normalTarget,1)).xyz-(uMMatrix*vec4(aPosition,1)).xyz;
     newNormal=normalize(newNormal);                         //对法向量规格化
     //计算从表面点到光源位置的向量 vp
     vec3 vp= normalize(lightLocation-(uMMatrix*vec4(aPosition,1)).xyz);
     vp=normalize(vp);                                       //规格化 vp
     float nDotViewPosition=max(0.0,dot(newNormal,vp));
                                                             //求法向量与 vp 向量的点积与 0 的最大值
     diffuse=lightDiffuse*nDotViewPosition;                  //计算散射光的最终强度
   }
   void main(){
      gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点的位置
      vec4 diffuseTemp=vec4(0.0,0.0,0.0,0.0);
     pointLight(normalize(aNormal), diffuseTemp, uLightLocation, vec4(0.8,0.8,0.8,1.0));
      vDiffuse=diffuseTemp;                             //将散射光最终强度传给片元着色器
      vPosition = aPosition;                            //将顶点的位置传给片元着色器
   }
```

• 第1-7行为顶点着色器中全局变量的声明，主要是增加了光源位置一致变量uLightLocation、法向量属性变量aNorma、以及用于传递给片元着色器的散射光最终强度易变变量vDiffuse。

• 第8-22行为根据前面介绍的公式计算散射光最终强度的pointLight方法，最重要的是在进行计算前要对顶点法向量进行变换，将法向量变换到当前的姿态下。

• 第23-29行为顶点着色器的main方法，其中首先增加了调用pointLight方法计算散射光最终强度的代码，同时也增加了将计算出的散射光最终强度值传递给片元着色器的代码。

（5）完成了顶点着色器的开发后，就可以开发片元着色器了，其代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_3/assets目录下的frag.sh。

```
1 precision mediump float;                 //给出默认浮点精度
2 uniform float uR;                        //球的半径
3 varying vec3 vPosition;                  //接收从顶点着色器传递过来的顶点位置
4 varying vec4 vDiffuse;                   //接收从顶点着色器传递过来的散射光最终强度
5 void main() {
6    vec3 color;
7    ……//此处省略了计算片元颜色值的代码，与前面案例中的相同
8    ……//请读者自行查看随书光盘中的源代码
9    vec4 finalColor=vec4(color,0);        //最终颜色
10      gl_FragColor=finalColor*vDiffuse;   //根据散射光最终强度计算片元的最终颜色值
11   }
```

• 第4行增加了vDiffuse易变变量的声明，其功能为接收从顶点着色器传递过来的散射光最终强度。

• 第10行在将片元颜色值传递给渲染管线之前，增加了其与散射光最终强度加权计算的操作。

6.2.4　镜面光

使用了上一小节中介绍的散射光效果后，场景的整体效果有了较大的提升。但这并不是光照的全部，现实世界中当光滑表面被照射时会有方向很集中的反射光。这就是镜面光（Specular），本小节将详细介绍镜面光的计算模型。

与散射光最终强度仅依赖于入射光与被照射点法向量的夹角不同，镜面光的最终强度还依赖于观察者的位置。也就是说，如果从摄像机到被照射点的向量不在反射光方向集中的范围内，观察者将不会看到镜面光，图6-12简单地说明了这个问题。

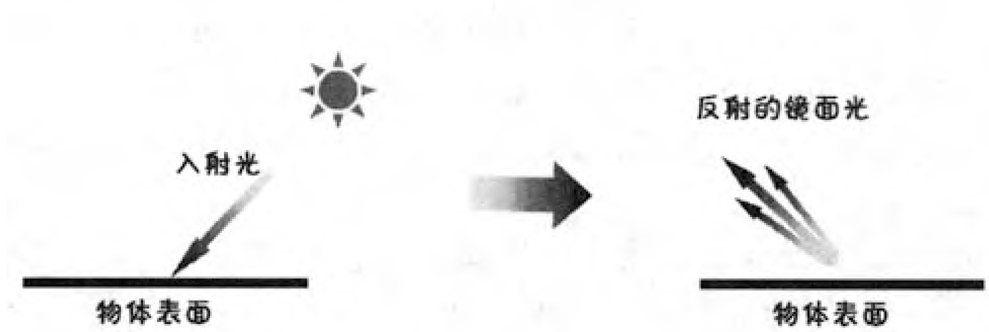


图6-12　镜面光基本情况

镜面光的计算模型比前面的两种光都要复杂一些，具体公式如下。

镜面光照射结果=材质的反射系数×镜面光强度×max（0,（cos（半向量与法向量的夹角））^{粗糙度} ）

实际开发中往往分两步进行计算，此时公式被拆解为如下情况。

镜面光最终强度=镜面光强度×max（0，（cos（半向量与法向量的夹角））^{粗糙度} ）

镜面光照射结果=材质的反射系数×镜面光最终强度

提示

材质的反射系数实际指的就是物体被照射处的颜色，镜面光强度指的是镜面光中RGB（红、绿、蓝）3个色彩通道的强度。

从上述公式中可以看出，与散射光计算公式主要有两点区别。首先是计算余弦值时对应的角不再是入射角，而是半向量与法向量的夹角。半向量指的是从被照射点到光源的向量与从被照射点到观察点向量的平均向量，图6-13说明了半向量的含义。

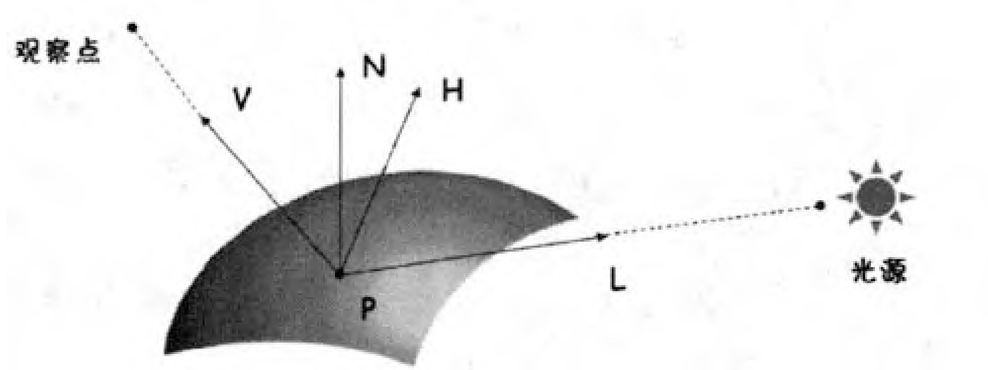


图6-13　计算镜面反射光

> **说明**
>
> 图6-13中V为从被照射点到观察点的向量，N为被照射点表面的法向量，H为半向量，L为从被照射点到光源的向量。

从图6-13中可以看出，半向量H与V及L共面，并且其与这两个向量的夹角相等。因此已知V和L后计算H非常简单，只要首先将V和L规格化，然后将规格化后的V与L求和并再次规格 化即可。求得半向量后，再求其与法向量夹角的余弦值就非常简单了，只需将规格化后的法向量与半向量进行点积即可。

另外一个区别就是求得的余弦值还需要对粗糙度进行乘方运算，此运算可以达到粗糙度越小，镜面光面积越大的效果，这也是很贴近现实世界的。

> **提示**
>
> 由于本书篇幅有限，故仅仅介绍了镜面光计算公式本身，而没有深入讨论为什么会产生这样的公式，有兴趣的读者可以参考其他相关资料或书籍。

了解了镜面光的基本原理后，下面给出一个使用镜面光的案例Sample6_4，其运行效果如图6-14所示。

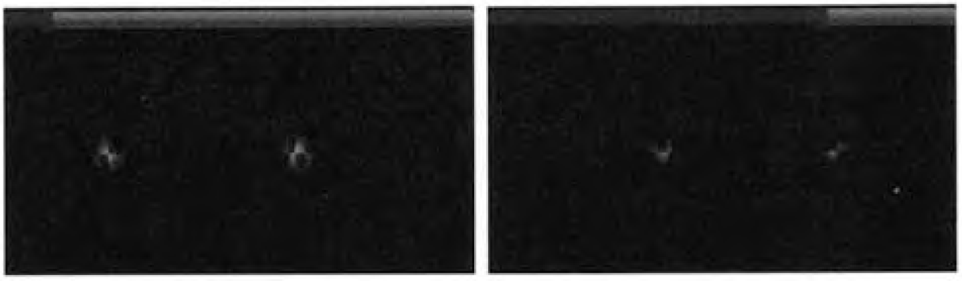


图6-14　案例Sample6_4的运行效果图

说明

图6-14中左侧为粗糙度值等于25的情况，右侧为粗糙度值等于50的情况。从左右两侧运行效果图的对比中可以看出，粗糙度越小，镜面光面积越大，这也符合我们观察现实世界的经验。另外从图中还可以看出，镜面光也是随光源位置的变化而变化的。

了解了镜面光的基本原理及案例的运行效果后，就可以进行案例的开发了。由于实际上本案例仅仅是将案例Sample6_3复制了一份并进行了修改，因此这里仅给出修改的主要步骤，具体如下所列。

（1）由于镜面光的计算不仅与光源位置有关，还与摄像机位置有关，故摄像机的位置也需要传入渲染管线。为了实现此目标，首先需要对MatrixState类中设置摄像机的方法进行升级并增加相关的成员变量，具体代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_4/com/bn/Sample6_4目录下的MatrixState.java。

```
1 static ByteBuffer llbb= ByteBuffer.allocateDirect(3*4);  //待用的字节缓冲
2 static float[] cameraLocation=new float[3];              //摄像机位置
3 public static FloatBuffer cameraFB;                      //摄像机位置数据缓冲
4 public static void setCamera(                            //设置摄像机的方法
5            float cx, float cy, float cz,                 //摄像机位置的 X、Y、Z 坐标
6            float tx, float ty, float tz,                 //观察目标点 X、Y、Z 坐标
7            float upx, float upy, float upz               //up 向量在 X、Y、Z 轴上的分量
8 ){
9    Matrix.setLookAtM(mVMatrix, 0, cx, cy, cz, tx, ty, tz, upx, upy, upz);
                                                            //产生摄像机观察矩阵
10       cameraLocation[0]=cx;cameraLocation[1]=cy;cameraLocation[2]=cz;
                                                            //将摄像机坐标记录进数组
11       llbb.clear();                                      //清除摄像机位置缓冲
12       llbb.order(ByteOrder.nativeOrder());               //设置字节顺序
13       cameraFB=llbb.asFloatBuffer();                     //转换为 float 型缓冲
14       cameraFB.put(cameraLocation);                      //将摄像机位置放入缓冲
15       cameraFB.position(0);                              //设置缓冲区起始位置
16   }
```

- 第1-3行为该类中新增加的成员变量，主要有摄像机位置数组、摄像机位置缓冲和待用的字节缓冲。
- 第4-16行为设置摄像机位置的方法，主要是增加了将摄像机X、Y、Z坐标先存储进数组，然后再存放进对应的float型缓冲的相关代码。摄像机位置存放进缓冲后，绘制时就可以根据需要传送进渲染管线供着色器进行镜面光计算使用。

（2）接着需要修改的是Ball类，主要是增加了将摄像机位置传送进渲染管线的相关代码，具体内容如下。

代码位置：见随书光盘中源代码/第6章/Sample6_4/com/bn/Sample6_4目录下的Ball.java。

```
1 public class Ball {
2     ……//此处省略了部分代码，请读者自行查看随书光盘中的源代码
3     int maCameraHandle;                                  //摄像机位置属性引用
4     public Ball(MySurfaceView mv) {/*代码省略*/}
5     public void initVertexData() {/*代码省略*/}
6     public void initShader(MySurfaceView mv) {           //初始化着色器
7         ……//此处省略了部分代码，请读者自行查看随书光盘中的源代码
8       maCameraHandle=GLES20.glGetUniformLocation(        //获取摄像机位置一致变量引用
9             mProgram, "uCamera");
10        }
11        public void drawSelf() {
12            ……//此处省略了部分代码，请读者自行查看随书光盘中的源代码
13          GLES20.glUniform3fv(                            //将摄像机位置传入渲染管线
14                maCameraHandle, 1, MatrixState.cameraFB);
15            ……//此处省略了部分代码，请读者自行查看随书光盘中的源代码
16            GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount); //绘制球
17    }}
```

- 第3行增加了声明摄像机位置一致变量引用的代码。
- 第8-9行在初始化着色器的方法中增加了获取摄像机位置一致变量引用的代码。
- 第11-17行为绘制球的drawSelf方法，其中增加了将摄像机位置传送进渲染管线的代码。

（3）完成了上述修改后，对Java代码的修改就基本完成了，下面需要修改的是着色器的代码。首先是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_4/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;  //总变换矩阵
uniform mat4 uMMatrix;          //变换矩阵
uniform vec3 uLightLocation;    //光源位置
uniform vec3 uCamera;           //摄像机位置
attribute vec3 aPosition;       //顶点位置
attribute vec3 aNormal;         //法向量
varying vec3 vPosition;         //用于传递给片元着色器的顶点位置
varying vec4 vSpecular;         //用于传递给片元着色器的镜面光最终强度
void pointLight(                //定位光光照计算的方法
    in vec3 normal,             //法向量
    inout vec4 specular,        //镜面光最终强度
    in vec3 lightLocation,      //光源位置
    in vec4 lightSpecular       //镜面光强度
  ){
    vec3 normalTarget=aPosition+normal;   //计算变换后的法向量
    vec3 newNormal=(uMMatrix*vec4(normalTarget,1)).xyz-(uMMatrix*vec4(aPosition,1)).xyz;
    newNormal=normalize(newNormal);        //对法向量规格化
    //计算从表面点到摄像机的向量
    vec3 eye= normalize(uCamera-(uMMatrix*vec4(aPosition,1)).xyz);
    //计算从表面点到光源位置的向量 vp
    vec3 vp= normalize(lightLocation-(uMMatrix*vec4(aPosition,1)).xyz);
    vp=normalize(vp);//格式化 vp
    vec3 halfVector=normalize(vp+eye);     //求视线与光线的半向量
    float shininess=50.0;                  //粗糙度，越小越光滑
    float nDotViewHalfVector=dot(newNormal,halfVector);         //法线与半向量的点积
    float powerFactor=max(0.0,pow(nDotViewHalfVector,shininess)); //镜面反射光强度因子
    specular=lightSpecular*powerFactor;   //最终的镜面光强度
  }
  void main()  {
    gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点的位置
    vec4 specularTemp=vec4(0.0,0.0,0.0,0.0);
    pointLight(normalize(aNormal), specularTemp, uLightLocation, vec4(0.7,0.7,0.7,
    1.0));                                //计算镜面光
    vSpecular=specularTemp;          /    /将最终镜面光强度传给片元着色器
    vPosition = aPosition;                //将顶点的位置传给片元着色器
  }
```

• 第1-8行为顶点着色器中全局变量的声明，主要是增加了摄像机位置一致变量uCamera、用于传递给片元着色器的镜面光最终强度易变变量vSpecular。

• 第9-28行为根据前面介绍的公式计算镜面光最终强度的pointLight方法。

• 第29-35行为顶点着色器的main方法，其中首先调用pointLight方法计算镜面光的最终强度，并将计算出的镜面光最终强度值传递给片元着色器。

（4）完成了顶点着色器的开发后，就可以开发片元着色器了，其代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_4/assets目录下的frag.sh。

```
precision mediump float;                    //给出默认的浮点精度
uniform float uR;
varying vec3 vPosition;                     //接收从顶点着色器传递过来的顶点位置
varying vec4 vSpecular;                     //接收从顶点着色器传递过来的镜面光最终强度
void main(){
  vec3 color;
  ……//此处省略了计算片元颜色值的代码，请读者自行查看随书光盘中的源代码
  vec4 finalColor=vec4(color,0);            //最终颜色
  gl_FragColor=finalColor*vDiffuse;         //根据镜面光最终强度计算片元的最终颜色值
}
```

说明

上述片元着色器的代码与散射光的套路基本一致，只是将接收散射光最终强度并与片元颜色值加权计算换成了接收镜面光最终强度并与片元颜色值加权计算。

6.2.5　三种光照通道的合成

前面3个小节案例中的每个仅采用了一种光照通道，而现实世界中3种通道是同时作用的。因此本小节将通过一个案例Sample6_5将前面3个小节不同通道（环境光、散射光、镜面光）的光照效果综合起来，其运行效果如图6-15所示。

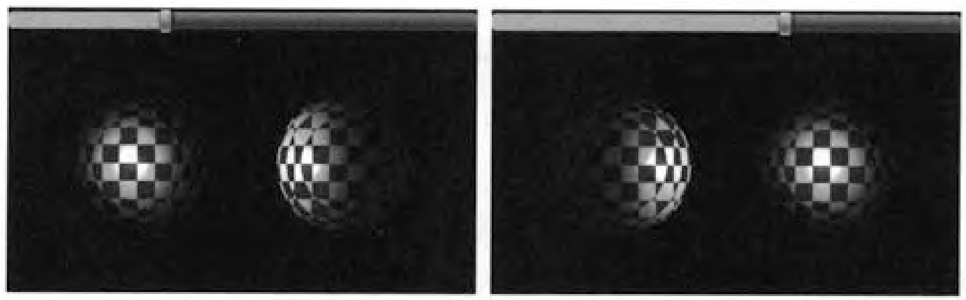


图6-15　案例Sample6_5的运行效果图

> **提示**
>
> 从图6-15中可以看出，综合了3种光照通道后，场景的真实感大大增强。

了解了案例的运行效果后，就可以进行案例的开发了。由于实际上本案例仅仅是将案例Sample6_2、Sample6_3和Sample6_4中顶点着色器及片元着色器计算光照的相关代码进行了综合，因此这里仅给出综合后着色器的代码。

（1）首先综合了3种光照通道的顶点着色器，其具体代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_5/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                //总变换矩阵
uniform mat4 uMMatrix;                  //变换矩阵
uniform vec3 uLightLocation;            //光源位置
uniform vec3 uCamera;                   //摄像机位置
attribute vec3 aPosition;               //顶点位置
attribute vec3 aNormal;                 //法向量
varying vec3 vPosition;                 //用于传递给片元着色器的顶点位置
varying vec4 vAmbient;                  //用于传递给片元着色器的环境光最终强度
varying vec4 vDiffuse;                  //用于传递给片元着色器的散射光最终强度
```

```
varying vec4 vSpecular;                    //用于传递给片元着色器的镜面光最终强度
void pointLight(                           //定位光光照计算的方法
  in vec3 normal,                          //法向量
  inout vec4 ambient,                      //环境光最终强度
  inout vec4 diffuse,                      //散射光最终强度
  inout vec4 specular,                     //镜面光最终强度
  in vec3 lightLocation,                   //光源位置
  in vec4 lightAmbient,                    //环境光强度
  in vec4 lightDiffuse,                    //散射光强度
  in vec4 lightSpecular                    //镜面光强度
){
  ambient=lightAmbient;                    //直接得出环境光的最终强度
  vec3 normalTarget=aPosition+normal;  //计算变换后的法向量
  vec3 newNormal=(uMMatrix*vec4(normalTarget,1)).xyz-(uMMatrix*vec4(aPosition,1)).xyz;
  newNormal=normalize(newNormal);          //对法向量规格化
  //计算从表面点到摄像机的向量
  vec3 eye= normalize(uCamera-(uMMatrix*vec4(aPosition,1)).xyz);
  //计算从表面点到光源位置的向量 vp
  vec3 vp= normalize(lightLocation-(uMMatrix*vec4(aPosition,1)).xyz);
  vp=normalize(vp);//格式化 vp
  vec3 halfVector=normalize(vp+eye);   //求视线与光线的半向量
  float shininess=50.0;                    //粗糙度，越小越光滑
  float nDotViewPosition=max(0.0,dot(newNormal,vp)); //求法向量与 vp 的点积与 0 的最大值
  diffuse=lightDiffuse*nDotViewPosition;               //计算散射光的最终强度
  float nDotViewHalfVector=dot(newNormal,halfVector);//法线与半向量的点积
  float powerFactor=max(0.0,pow(nDotViewHalfVector,shininess)); //镜面反射光强度因子
  specular=lightSpecular*powerFactor;                  //计算镜面光的最终强度
}
void main(){
   gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点位置
   vec4 ambientTemp,diffuseTemp,specularTemp;    //用来接收 3 个通道最终强度的变量
pointLight(normalize(aNormal),ambientTemp,diffuseTemp,specularTemp,uLightLocation,
   vec4(0.15,0.15,0.15,1.0),vec4(0.8,0.8,0.8,1.0),vec4(0.7,0.7,0.7,1.0));
                                   //计算定位光各通道强度
   vAmbient=ambientTemp;           //将环境光最终强度传给片元着色器
   vDiffuse=diffuseTemp;           //将散射光最终强度传给片元着色器
   vSpecular=specularTemp;         //将镜面光最终强度传给片元着色器
   vPosition = aPosition;          //将顶点的位置传给片元着色器
}
```

提示

上述代码只是将3种通道光照的计算都综合到了pointLight方法中，并将计算出来的3种通道光的最终强度都传递给了片元着色器。

（2）完成了顶点着色器的开发后，就可以开发片元着色器了，其代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_5/assets目录下的frag.sh。

```
precision mediump float;                      //指定默认浮点精度
uniform float uR;                             //球的半径
varying vec3 vPosition;                       //接收从顶点着色器传递过来的顶点位置
varying vec4 vAmbient;                        //接收从顶点着色器传递过来的环境光最终强度
varying vec4 vDiffuse;                        //接收从顶点着色器传递过来的散射光最终强度
varying vec4 vSpecular;                       //接收从顶点着色器传递过来的镜面反射光最终强度
void main(){
  vec3 color;
  ……//此处省略了计算片元颜色值的代码，请读者自行查看随书光盘中的源代码
     vec4 finalColor=vec4(color,0);       //最终颜色
     //综合3个通道光的最终强度及片元的颜色计算出最终片元的颜色并传递给管线
     gl_FragColor=finalColor*vAmbient + finalColor*vDiffuse + finalColor*vSpecular;
   }
```

提示

上述代码将原来单独计算的各个光照通道综合在一起计算，产生同时受3个通道影响的最终片元颜色值后再将其传递给渲染管线。

前面的案例中都是将光各个通道的原始强度固化在着色器程序中的，如果有需要，读者可以自行开发出将光各个通道的强度由宿主程序传入着色器一致变量的版本。另外前面的案例中也仅有一个光源，如果有需要，读者可以根据前面案例介绍的内容开发出多个光源叠加照射的效果。

还需要注意的是，本节介绍的光照计算模型是比较常用的也是比较简单的一套，还有很多其他更为复杂的可以取得更好效果的光照计算模型，读者有需要也可以进一步参考其他技术资料。

6.3　定位光与定向光

上一节中介绍的光照效果都是基于定位光光源的，定位光光源类似于现实生活中的白炽灯灯泡，其在某个固定的位置，发出的光向四周发散。定位光照射的一个明显特点就是，在给定光源位置的情况下，对不同位置的物体产生的光照效果不同。

现实世界中并不都是定位光，例如照射到地面上的阳光，光线之间是平行的，这种光称之为定向光。定向光照射的明显特点是，在给定光线方向的情况下，场景中不同位置的物体反映出的光照效果完全一致。图6-16中对定位光与定向光的照射特点进行了比较。

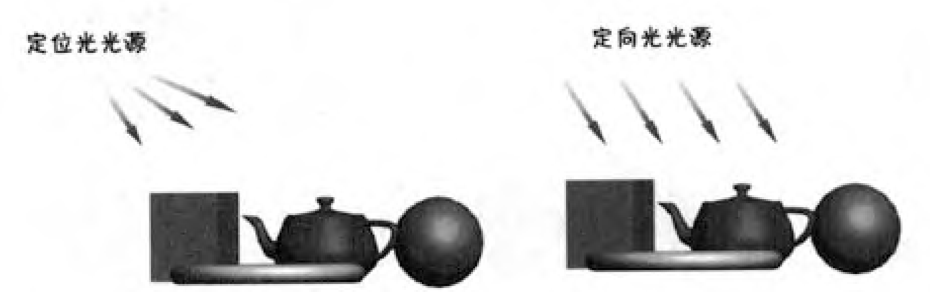


图6-16　定位光和定向光

了解了定位光与定向光的区别后，接下来将通过一个案例Sample6_6介绍定向光效果的开发，其运行效果如图6-17所示。

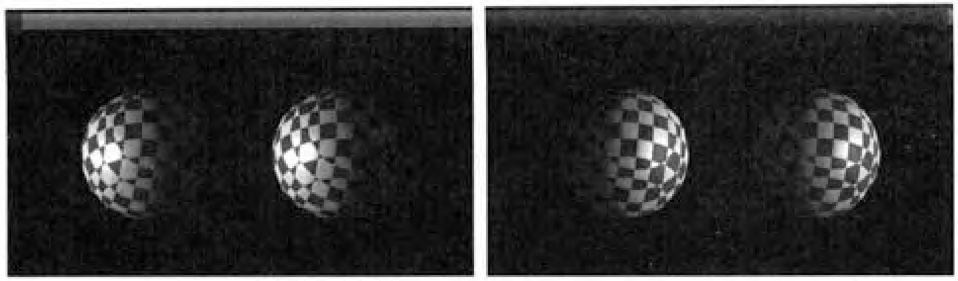


图6-17　案例Sample6_6的运行效果图

说明

图6-17中左侧的图表示定向光方向从左向右照射的情况，右侧的图表示定向光方向从右向左照射的情况。从左右两幅效果图的对比中可以看出，在定向光方向确定的情况下，其对场景中任何位置的物体都产生相同的光照效果。

了解了定向光的基本原理及案例的运行效果后，就可以进行案例的开发了。由于实际上本案例仅仅是将案例Sample6_5复制了一份并进行了修改，因此这里仅给出修改的主要步骤，具体如下所列。

（1）由于使用定向光需要将光线的方向由Java程序传入渲染管线，为了方便首先需要对MatrixState类进一步升级，在其中增加设置定向光方向并存入缓冲的相关代码，具体情况如下。

代码位置：见随书光盘中源代码/第6章/Sample6_6/com/bn/Sample6_6目录下的MatrixState.java。

```
1 public static float[] lightDirection = new float[] { 0, 0, 1 };  //定向光方向向量数组
2 public static FloatBuffer lightDirectionFB;                       //定向光方向向量数据缓冲
3 public static void setLightDirection(float x, float y, float z) { //设置定向光方向的方法
4     llbbL.clear();                                               //清空缓冲
5     //将定向光方向向量的X、Y、Z分量存入数组
6     lightDirection[0] = x;lightDirection[1] = y;lightDirection[2] = z;
7     llbbL.order(ByteOrder.nativeOrder());                        //设置字节顺序
8     lightDirectionFB = llbbL.asFloatBuffer();                    //转换为float型缓冲
9     lightDirectionFB.put(lightDirection);                        //将光源方向向量放入数据缓冲
10        lightDirectionFB.position(0);                            //设置缓冲区起始位置
11    }
```

- 第1-2行为该类中新增加的成员变量，包括定向光方向向量数组lightDirection、定向光方向向量数据缓冲lightDirectionFB。
- 第3-11行为设置定向光方向的方法，该方法首先将接收的定向光方向向量X、Y、Z轴的分量依次存入数组，然后再将数组中的值存入缓冲，供绘制时传入渲染管线。

（2）接着需要修改的是Ball类，在该类中增加了将定向光方向传入渲染管线的相关代码，具体情况如下。

代码位置：见随书光盘中源代码/第6章/Sample6_6/com/bn/Sample6_6目录下的Ball.java。

```
public class Ball {
  ……//此处省略了部分代码，请读者自行查看随书光盘中的源代码
  int muLightDirectionHandle;                          //定向光方向一致变量的引用
```

```
4 public Ball(MySurfaceView mv) {/*代码省略*/}
5 public void initVertexData() {/*代码省略*/}
6 public void initShader(MySurfaceView mv) {    //初始化着色器
7    ……//此处省略了部分代码，请读者自行查看随书光盘中的源代码
8    muLightDirectionHandle=GLES20.glGetUniformLocation(
9        mProgram, "uLightDirection");        //获取着色器中定向光方向一致变量的引用
10   }
11      public void drawSelf() {
12          ……//此处省略了部分代码，请读者自行查看随书光盘中的源代码
13          GLES20.glUniform3fv(              //将定向光方向传入渲染管线
14              muLightDirectionHandle, 1, MatrixState.lightDirectionFB);
15          ……//此处省略了部分代码，请读者自行查看随书光盘中的源代码
16          GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount); //绘制球体
17 }}
```

- 第3行增加了声明定向光方向一致变量引用muLightDirectionHandle的代码。
- 第8-9行在初始化着色器的方法中增加了获取定向光方向一致变量引用值的代码。
- 第11-17行为绘制球的drawSelf方法，其中增加了将定向光方向传送进渲染管线的代码。

（3）完成了Ball类的修改后，对Java代码的修改就基本完成了，下面就可以修改着色器的代码了。由于本案例中的片元着色器与Sample6_5中的相同，因此这里仅介绍顶点着色器，其具体代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_6/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                    //总变换矩阵
uniform mat4 uMMatrix;                      //变换矩阵
uniform vec3 uLightDirection;               //定向光方向
uniform vec3 uCamera;                       //摄像机位置
attribute vec3 aPosition;                   //顶点位置
attribute vec3 aNormal;                     //法向量
varying vec3 vPosition;                     //用于传递给片元着色器的顶点位置
varying vec4 vAmbient;                      //用于传递给片元着色器的环境光最终强度
varying vec4 vDiffuse;                      //用于传递给片元着色器的散射光最终强度
   varying vec4 vSpecular;                  //用于传递给片元着色器的镜面光最终强度
   void directionalLight(                   //定向光光照计算的方法
     in vec3 normal,                        //法向量
     inout vec4 ambient,                    //环境光最终强度
     inout vec4 diffuse,                    //散射光最终强度
     inout vec4 specular,                   //镜面光最终强度
     in vec3 lightDirection,                //定向光方向
     in vec4 lightAmbient,                  //环境光强度
     in vec4 lightDiffuse,                  //散射光强度
     in vec4 lightSpecular                  //镜面光强度
   ){
     ambient=lightAmbient;                  //直接得出环境光的最终强度
     vec3 normalTarget=aPosition+normal;    //计算变换后的法向量
     vec3 newNormal=(uMMatrix*vec4(normalTarget,1)).xyz-(uMMatrix*vec4(aPosition,1)).xyz;
```

```
24      newNormal=normalize(newNormal);  //对法向量规格化
25      //计算从表面点到摄像机的向量
26      vec3 eye= normalize(uCamera-(uMMatrix*vec4(aPosition,1)).xyz);
27      vec3 vp= normalize(lightDirection);  //规格化定向光方向向量
28      vec3 halfVector=normalize(vp+eye);    //求视线与光线的半向量
29      float shininess=50.0;                 //粗糙度，越小越光滑
30      float nDotViewPosition=max(0.0,dot(newNormal,vp)); //求法向量与 vp 的点积与 0 的最大值
31      diffuse=lightDiffuse*nDotViewPosition;               //计算散射光的最终强度
32      float nDotViewHalfVector=dot(newNormal,halfVector);    //法线与半向量的点积
33      float powerFactor=max(0.0,pow(nDotViewHalfVector,shininess)); //镜面反射光强度因子
34      specular=lightSpecular*powerFactor;           //计算镜面光的最终强度
35    }
36    void main(){
37       gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点位置
38       vec4 ambientTemp,diffuseTemp,specularTemp;    //用来接收 3 个通道最终强度的变量
39
directionalLight(normalize(aNormal),ambientTemp,diffuseTemp,specularTemp,uLightDirect
ion,
40       vec4(0.15,0.15,0.15,1.0),vec4(0.8,0.8,0.8,1.0),vec4(0.7,0.7,0.7,1.0));
                                      //计算定向光各通道强度
41       vAmbient=ambientTemp;        //将环境光最终强度传给片元着色器
42       vDiffuse=diffuseTemp;        //将散射光最终强度传给片元着色器
43       vSpecular=specularTemp;      //将镜面光最终强度传给片元着色器
44       vPosition = aPosition;       //将顶点的位置传给片元着色器
45    }
```

- 第3行将原来定位光光源位置一致变量的声明替换成了定向光方向向量一致变量的声明。

• 第11-35行将原来计算定位光光照的pointLight方法替换成了计算定向光光照的directionalLight方法。上述两个方法主要有两点不同，首先是原来表示定位光光源位置的参数lightLocation被换成了表示定向光方向的参数lightDirection。另一个是计算时所需的光方向向量直接从lightDirection参数规格化即可得到，不需要再通过光源位置与被照射点位置进行计算了。

> **提示**
>
> 从上述顶点着色器的代码中可以看出，与定位光相同，定向光也是分环境光、散射光、镜面光3个通道进行计算的。

6.4 点法向量和面法向量

本章前面几节的案例都是基于球面开发的，球面属于连续、平滑的曲面，因此面上的每个顶点都有确定的法向量。但现实世界中的物体表面并不都是连续、平滑的，此时对于面上的某些点的法向量计算就不那么直观了，图6-18说明了这个问题。

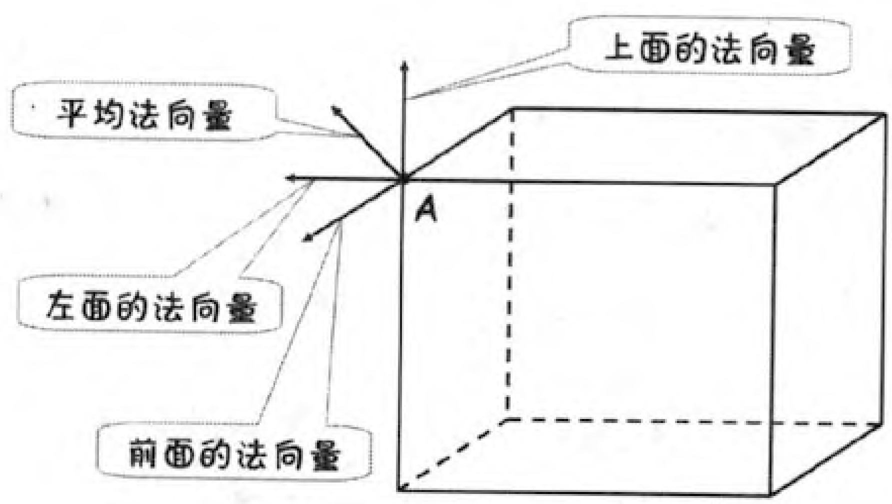


图6-18 两种法向量示意图

从图6-18中可以看出，顶点A位于长方体左、上、前3个面的交界处，此处是不光滑的。这种情况下顶点 A的法向量有两种处理策略，

具体如下所列。

- 在顶点A的位置放置3个不同的顶点，每个顶点看作是仅属于一个面。各个顶点的法向量即为其属于的面的法向量，这种策略就是面法向量的策略，比较适合棱角分明的物体。
- 顶点A的位置仅认为存在一个顶点，其法向量取其所属的所有面法向量的平均值。这种策略就是点法向量策略，比较适合用多个平面搭建平滑曲面的情况。

> **提示**
>
> 前面球体的案例采用的就是此策略，只不过由于球面上顶点的法向量可以直接计算，因此相当于直接得到了点平均法向量，而略去了计算平均值的过程。但很多情况下是需要进行平均法向量的计算的，尤其是在加载预制3D模型的时候，这一点在后面的章节会有专门的介绍。

了解了点法向量和面法向量的基本知识后，下面将通过两个基本相同的绘制立方体的案例（Sample6_7和Sample6_8）对这两种策略进行比较。这两个案例的运行效果如图6-19和图6-20所示，其中图6-19来自采用面法向量策略的案例Sample6_7，图6-20来自采用点法向量策略的案例Sample6_8。

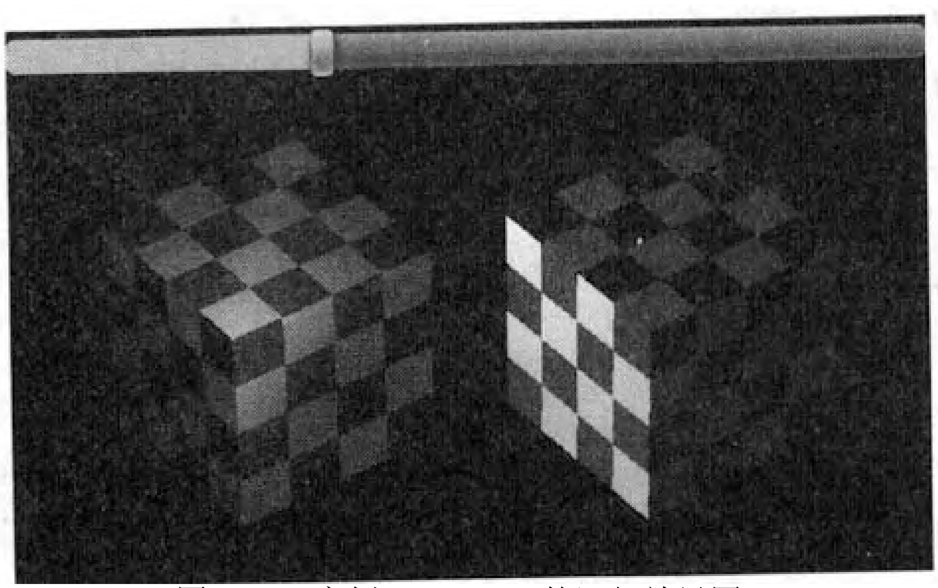


图6-19　案例Sample6_7的运行效果图

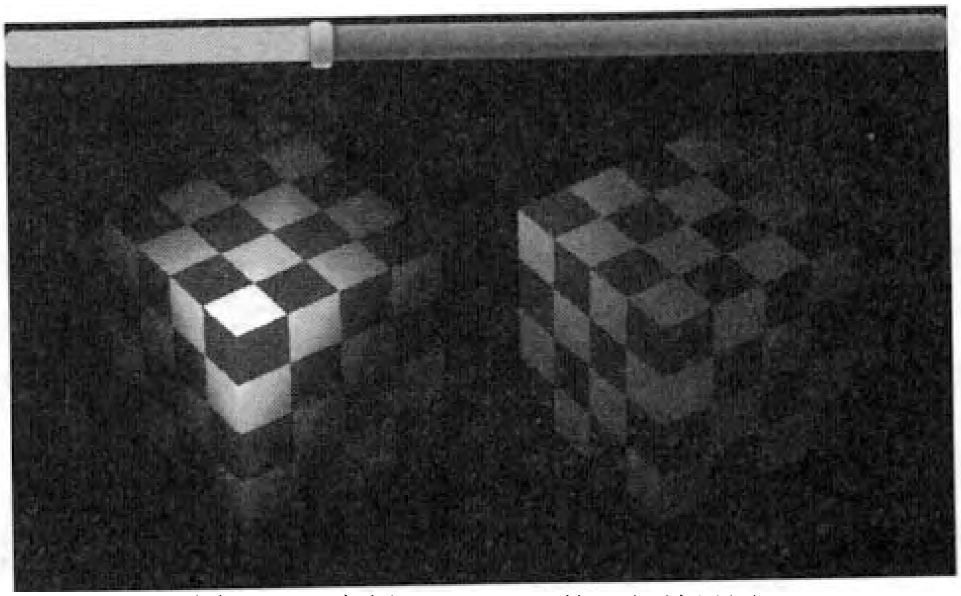


图6-20　案例Sample6_8的运行效果图

说明

从图6-19与图6-20的比较中可以看出，对于棱角分明的物体适合采用面法向量策略。若采用点法向量策略渲染真实感就会差很多了，实际开发中读者应该根据所绘制物体表面的特点来选用合适的策略。

了解了两个案例的运行效果后，就可以进行案例的开发了。由于这两个案例中大部分的代码与前面6.2.5小节中案例Sample6_5里的完全相同，主要的区别就在立方体顶点与法向量初始化的部分，所以下面仅给出Sample6_7与Sample6_8中初始化立方体顶点与法向量的部分代码，具体内容如下所列。

（1）首先介绍采用面法向量策略的案例Sample6_7中初始化立方体顶点及法向量数据的initVertexData方法，其代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_7/com/bn/Sample6_7目录下的Cube.java。

```
public void initVertexData() {           //初始化顶点数据的方法
    vCount=6*6;                          //顶点数(6个面，每个面两个三角形，6个顶点)
```

```
float vertices[]=new float[]{
    Constant.UNIT_SIZE,Constant.UNIT_SIZE,Constant.UNIT_SIZE,     //立方体前面
    -Constant.UNIT_SIZE,Constant.UNIT_SIZE,Constant.UNIT_SIZE,    //两个三角形
    -Constant.UNIT_SIZE,-Constant.UNIT_SIZE,Constant.UNIT_SIZE,   //中 6 个顶点
    Constant.UNIT_SIZE,Constant.UNIT_SIZE,Constant.UNIT_SIZE,     //的 X、Y、Z 坐标
    -Constant.UNIT_SIZE,-Constant.UNIT_SIZE,Constant.UNIT_SIZE,
    Constant.UNIT_SIZE,-Constant.UNIT_SIZE,Constant.UNIT_SIZE,
        ……//此处省略了其他 5 个面顶点坐标产生的代码，
        ……//需要的读者请自行查看随书光盘中的源代码
    };
    ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);
                                                    //创建顶点坐标数据缓冲
    vbb.order(ByteOrder.nativeOrder());             //设置字节顺序
    mVertexBuffer = vbb.asFloatBuffer();            //转换为 float 型缓冲
    mVertexBuffer.put(vertices);                    //向缓冲区中放入顶点坐标数据
    mVertexBuffer.position(0);                      //设置缓冲区起始位置
    float normals[]=new float[]{
        0,0,1,  0,0,1,  0,0,1,  0,0,1,  0,0,1,  0,0,1,    //前面上 6 个顶点的法向量
        0,0,-1,  0,0,-1,  0,0,-1,  0,0,-1,  0,0,-1,  0,0,-1, //后面上 6 个顶点的法向量
        -1,0,0,  -1,0,0,  -1,0,0,  -1,0,0,  -1,0,0,  -1,0,0, //左面上 6 个顶点的法向量
        1,0,0,  1,0,0,  1,0,0,  1,0,0,  1,0,0,  1,0,0,    //右面上 6 个顶点的法向量
        0,1,0,  0,1,0,  0,1,0,  0,1,0,  0,1,0,  0,1,0,    //上面上 6 个顶点的法向量
        0,-1,0,  0,-1,0,  0,-1,0,  0,-1,0,  0,-1,0,  0,-1,0, //下面上 6 个顶点的法向量
    };
    ByteBuffer nbb = ByteBuffer.allocateDirect(normals.length*4);
                                                    //创建绘制顶点法向量缓冲
    nbb.order(ByteOrder.nativeOrder());             //设置字节顺序
    mNormalBuffer = nbb.asFloatBuffer();            //转换为 float 型缓冲
    mNormalBuffer.put(normals);                     //向缓冲区中放入顶点法向量数据
    mNormalBuffer.position(0);                      //设置缓冲区起始位置
}
```

- 第3-17行为立方体各顶点位置数据的初始化。
- 第17-30行为立方体各顶点法向量数据的初始化，其中给每个顶点设置了其所属面的法向量。

（2）接着介绍采用点法向量策略的案例Sample6_8中初始化立方体顶点及法向量数据的initVertexData方法，其代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_8/com/bn/Sample6_8目录下的Cube.java。

```
public void initVertexData() {                              //初始化顶点数据的方法
    vCount = 6 * 6;                                         //顶点数
    float vertices[] = new float[] {
        ……//此处省略了初始化立方体6个面各个顶点坐标数据的代码，
        ……//需要的读者请自行查看随书光盘中的源代码
    };
    ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);
                                                            //创建顶点坐标数据缓冲
    vbb.order(ByteOrder.nativeOrder());                     //设置字节顺序
    mVertexBuffer = vbb.asFloatBuffer();                    //转换为float型缓冲
```

```
    mVertexBuffer.put(vertices);                          //向缓冲区中放入顶点坐标数据
    mVertexBuffer.position(0);                            //设置缓冲区起始位置
    float normals[] = vertices;                           //顶点法向量数据
    ByteBuffer nbb = ByteBuffer.allocateDirect(normals.length * 4);
                                                          //创建绘制顶点法向量缓冲
    nbb.order(ByteOrder.nativeOrder());                   //设置字节顺序
    mNormalBuffer = nbb.asFloatBuffer();                  //转换为float型缓冲
    mNormalBuffer.put(normals);                           //向缓冲区中放入顶点法向量数据
    mNormalBuffer.position(0);                            //设置缓冲区起始位置
}
```

提示

由于本案例中原始情况下将立方体的几何中心放在了坐标原点，因此每个顶点的平均法向量就没有必要真正进行求和再平均的计算了，直接采用顶点的X、Y、Z坐标代替即可。但开发中并不总是像本章的案例一样可以进行简化，后面的章节会给出需要详细计算的案例。

6.5　光照的每顶点计算与每片元计算

细心的读者会发现，本章前面的案例都是在顶点着色器中进行光照计算的。这是由于在顶点着色器中对每个顶点进行光照计算后得到顶点的最终光照强度，再由管线插值后传入片元着色器以计算片元的颜色，这样一方面效率比较高，另一方面产生的光照效果也不错。

但由于这种计算方式插值的是基于顶点计算后的光照强度，因此在要求很高，希望有非常细腻光照效果的场合下就略显粗糙了。本节将介绍另一种光照计算方式，其首先将插值后的法向量数据传入片元

着色器，然后在片元着色器中进行光照计算。这种新的方式也称为每片元光照，可以取得为更细腻的光照效果。

进行案例开发之前需要首先了解一下本节两个案例（Sample6_9和Sample6_10）的运行效果，具体情况如图6-21所示。

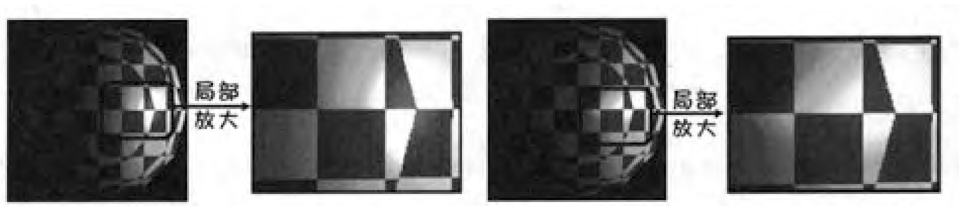


图6-21　案例Sample6_9和Sample6_10的运行效果对比图

说明

图6-21中左侧是每片元计算一次光照的案例Sample6_9的运行效果，右侧是每顶点计算一次光照的案例Sample6_10的运行效果。从两幅图的对比中可以看出，每片元执行一次光照使过渡更平滑，没有明显的边缘。另外，仅从图上观察可能区别还不是很明显，笔者建议读者用真机运行一下两个案例，将光源设置在不同的位置观察比较，区别会更明显。

了解了两个案例的运行效果后，就可以进行开发了。实际上这两个案例主要是将前面6.2.5小节中的案例Sample6_5复制并修改了部分代码而成的。其中Sample6_10仅修改了Java代码中切割球面的角度以及绘制球体的次数，没有本质变化，这里不再赘述，需要的读者请参考随书光盘中的源代码。

而案例Sample6_9除了也进行了Sample6_10的Java代码改动外，还大面积修改了顶点着色器与片元着色器，具体情况如下所列。

（1）首先介绍Sample6_9中修改后的顶点着色器，其具体代码如下。

代码位置：见随书光盘中源代码/第6章/Sample6_9/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                        //总变换矩阵
attribute vec3 aPosition;                       //顶点位置
attribute vec3 aNormal;                         //法向量
varying vec3 vPosition;                         //用于传递给片元着色器的顶点位置
varying vec3 vNormal;                           //用于传递给片元着色器的法向量
void main(){
   gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点位置
   vPosition = aPosition;                       //将顶点的位置传给片元着色器
   vNormal = aNormal;                           //将法向量传给片元着色器
  }
```

提示

从上述代码中可以看出，顶点着色器比改动前简单多了，没有了计算光照的大量代码，同时增加了将法向量通过易变变量传入片元着色器的代码。

（2）介绍完顶点着色器后，接着就应该介绍改动后的片元着色器了，其具体代码如下。

```
precision mediump float;                     //给出默认浮点精度
uniform float uR;                            //球的半径
uniform vec3 uLightLocation;                 //光源位置
uniform mat4 uMMatrix;                       //变换矩阵
uniform vec3 uCamera;                        //摄像机位置
varying vec3 vPosition;                      //接收从顶点着色器传递过来的顶点位置
varying vec3 vNormal;                        //接收从顶点着色器传递过来的法向量
void pointLight(                             //定位光光照计算的方法
  in vec3 normal,                            //法向量
      inout vec4 ambient,                    //环境光最终强度
      inout vec4 diffuse,                    //散射光最终强度
      inout vec4 specular,                   //镜面光最终强度
      in vec3 lightLocation,                 //光源位置
      in vec4 lightAmbient,                  //环境光强度
      in vec4 lightDiffuse,                  //散射光强度
      in vec4 lightSpecular                  //镜面光强度
    ){
      ambient=lightAmbient;                  //直接得出环境光的最终强度
      vec3 normalTarget=vPosition+normal;    //计算变换后的法向量
```

```
    vec3 newNormal=(uMMatrix*vec4(normalTarget,1)).xyz-(uMMatrix*vec4(vPosition,1)).xyz;
    newNormal=normalize(newNormal);  //对法向量规格化
    //计算从表面点到摄像机的向量
    vec3 eye= normalize(uCamera-(uMMatrix*vec4(vPosition,1)).xyz);
    //计算从表面点到光源位置的向量 vp
    vec3 vp= normalize(lightLocation-(uMMatrix*vec4(vPosition,1)).xyz);
    vp=normalize(vp);//格式化 vp
    vec3 halfVector=normalize(vp+eye);    //求视线与光线的半向量
    float shininess=50.0;                 //粗糙度，越小越光滑
    float nDotViewPosition=max(0.0,dot(newNormal,vp));//求法向量与 vp 的点积与 0 的最大值
    diffuse=lightDiffuse*nDotViewPosition;              //计算散射光的最终强度
    float nDotViewHalfVector=dot(newNormal,halfVector);    //法线与半向量的点积
    float powerFactor=max(0.0,pow(nDotViewHalfVector,shininess)); //镜面反射光强度因子
    specular=lightSpecular*powerFactor;             //计算镜面光的最终强度
  }
  void main() {
    ……//此处省略了计算片元颜色值的代码，请读者自行查看随书光盘中的源代码
    vec4 ambient,diffuse,specular;    //用来接收 3 个通道最终强度的变量
    pointLight(normalize(vNormal),ambient,diffuse,specular,uLightLocation,
                                      //计算定位光各通道强度
    vec4(0.15,0.15,0.15,1.0),vec4(0.8,0.8,0.8,1.0),vec4(0.7,0.7,0.7,1.0));
    //综合 3 个通道光的最终强度及片元的颜色计算出最终片元的颜色并传递给管线
    gl_FragColor=finalColor*ambient + finalColor*diffuse + finalColor*specular;
  }
```

提示

读者应该发现，上述片元着色器中的很多代码都是本章前面案例中多次出现过的，只不过前面的案例中都是在顶点着色器中出

现，而这里挪到了片元着色器中。因此每片元计算光照与每顶点计算光照算法并没有本质区别，只是代码执行的位置不同、效果与效率不同而已。实际开发中读者应该权衡速度、效果的要求，选用合适的计算策略。

6.6　本章小结

本章主要向读者介绍了OpenGL ES 2.0中光照的基本知识，掌握了本章所介绍的技术后，应该能开发出更加逼真的3D场景。

提示

读者可能会发现，本章中光照的案例都是没有阴影的，而阴影对于增加场景的真实感是非常重要的。是的，基本光照是不会自动产生阴影的，实际上阴影是3D开发中一个比较高级的话题，后面的章节会进行专门的讨论。

第7章　纹理映射

前面的章节已经介绍了变换、光照等方面的知识，通过这些知识已经可以渲染出具有一定真实感的场景。但到目前为止场景中物体的颜色是比较单一的，相比于绚丽多彩的现实世界显得有些乏味，因此仅掌握上述技术是远远不够的。

想要绘制出更加真实、酷炫的3D物体，就需要用到纹理映射。本章将对纹理映射方面的知识进行详细介绍，主要包括纹理映射的基本原理、两种不同的拉伸方式、两种不同的采样方式、MipMap纹理、多重纹理与过程纹理等。

7.1　初识纹理映射

本节将首先向读者介绍纹理映射的基本原理，然后通过一个三角形纹理映射的简单案例来介绍2D纹理是如何映射到3D场景中的立体物体上的。

7.1.1　基本原理

启用纹理映射功能后，如果想把一幅纹理应用到相应的几何图元，就必须告知渲染系统如何进行纹理的映射。告知的方式就是为图元中的顶点指定恰当的纹理坐标，纹理坐标用浮点数来表示，范围一般从0.0到1.0，下面的图7-1就给出了纹理映射的基本原理。

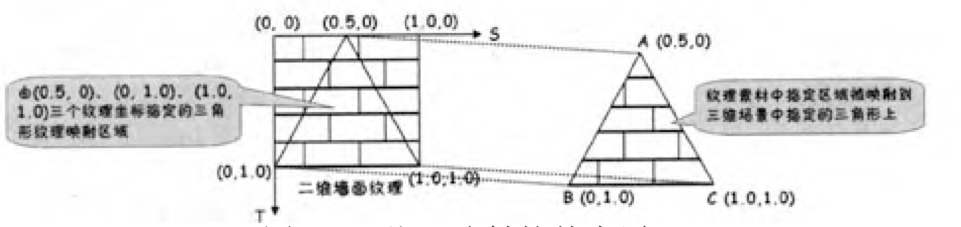


图7-1　纹理映射的基本原理

从图7-1中可以看出。

- 左侧是一幅纹理图，其位于纹理坐标系中。纹理坐标系原点在左上侧，向右为S轴，向下为T轴，两个轴的取值范围都是0.0～1.0。也就是说不论实际纹理图的尺寸如何，其横向、纵向坐标最大值都是1。若实际图为512像素×256像素，则横边第512个像素对应纹理坐标为1，竖边第 256个像素对应纹理坐标为1，其他情况依此类推。
- 右侧是一个三角形图元，其3个顶点A、B、C都指定了纹理坐标，3组纹理坐标正好在右侧的纹理图中确定了需要映射的三角形纹理区域。

从上述两点可以看出，纹理映射的基本思想就是首先为图元中的每个顶点指定恰当的纹理坐标，然后通过纹理坐标在纹理图中可以确定选中的纹理区域，最后将选中纹理区域中的内容根据纹理坐标映射到指定的图元上。

回忆一下第3章介绍过的渲染管线，最终用户看到的是显示在屏幕上的像素，而像素是由片元产生的。因此，进行纹理映射的过程实际上就是为右侧三角形图元中的每个片元着色，用于着色的颜色需要从左侧的纹理图中提取，具体过程如下。

- 首先图元中的每个顶点都需要在顶点着色器中通过易变变量将纹理坐标传入片元着色器。
- 经过顶点着色器后渲染管线的固定功能部分会根据情况进行插值计算，产生对应到每个片元的用于记录纹理坐标的易变变量值。
- 最后每个片元在片元着色器中根据其接收到的记录纹理坐标的易变变量值到纹理图中提取出对应位置的颜色即可，提取颜色的过程一般称之为纹理采样。

提示

OpenGL ES中进行纹理映射时，规定纹理图片尺寸的宽与高必须为2的n次方，如：8×8、16×8、32×32，64×64、32×256等。不满足2的n次方尺寸要求的纹理图在程序运行时会产生问题，请读者在开发中要特别注意这一点。

7.1.2　纹理映射的简单案例

介绍了纹理映射的基本原理后，本小节将给出一个将砖墙纹理映射到3D空间中三角形的案例。本案例中采用的原始纹理如图7-2所示，案例的具体运行效果如图7-3所示。

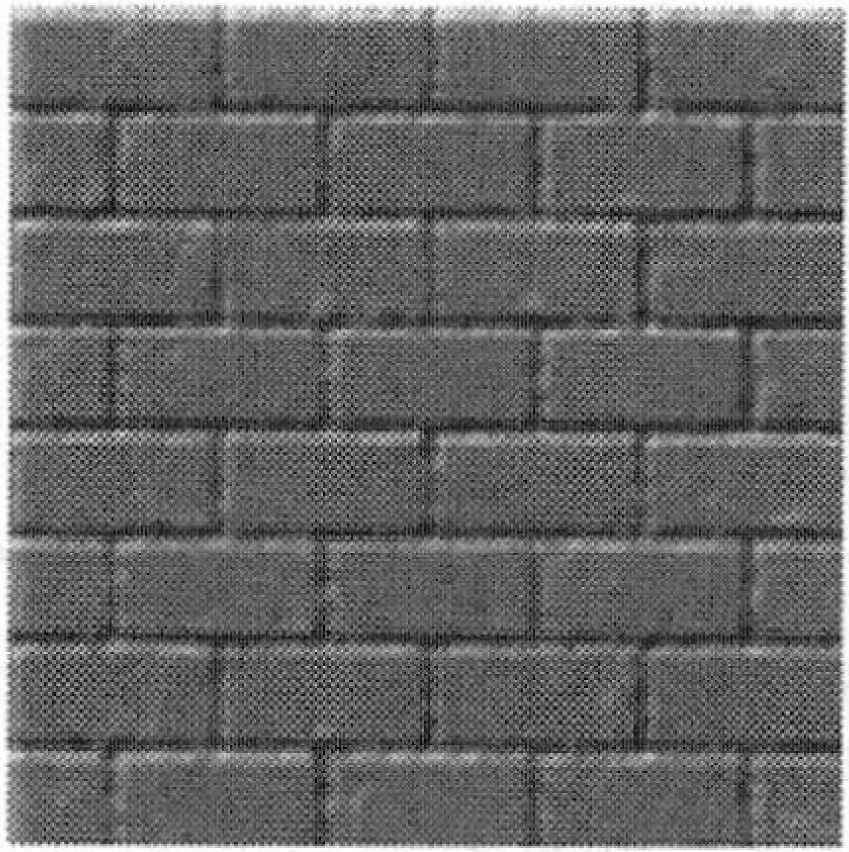


图7-2　原始纹理

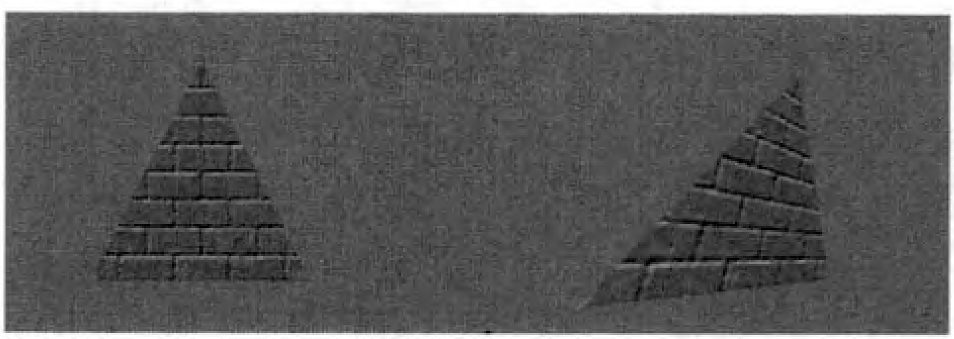


图7-3　案例运行的效果图

说明

图7-3中三角形的上面、左下、右下3个顶点的纹理坐标分别为（0.5,0）、（0,1）、（1,1），左侧的图是三角形原始姿态的情

况，右侧的是三角形旋转一定角度后的情况。

前面已经介绍了本案例的运行效果，接下来将介绍本案例的具体开发步骤。

（1）首先开发的是用于绘制场景的MySurfaceView类，其代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_1/com/bn/Sample7_1目录下的MySurfaceView.java。

```
package com.bn.Sample7_1;                                    //声明包
import java.io.IOException;                                  //引入相关类
```

```
......//此处省略部分类引入的代码，请读者自行查看随书光盘的源代码
import android.graphics.BitmapFactory;                    //引入相关类
class MySurfaceView extends GLSurfaceView {
  private final float TOUCH_SCALE_FACTOR = 180.0f/320;    //角度缩放比例
  private SceneRenderer mRenderer;                        //场景渲染器
  private float mPreviousY;                               //上次的触控位置 Y 坐标
  private float mPreviousX;                               //上次的触控位置 X 坐标
      int textureId;                                      //系统分配的纹理 id
      public MySurfaceView(Context context) {/*代码省略*/ }
      @Override                                           //触摸事件回调方法
      public boolean onTouchEvent(MotionEvent e) {  /*代码省略*/  }
      private class SceneRenderer implements GLSurfaceView.Renderer{
        Triangle texRect;                                 //纹理三角形对象引用
        public void onDrawFrame(GL10 gl) {
          GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT |    //清除深度缓冲与颜色缓冲
          GLES20.GL_COLOR_BUFFER_BIT);
          texRect.drawSelf(textureId);                    //绘制纹理矩形
        }
        public void onSurfaceChanged(GL10 gl, int width, int height) {*/代码省略*/}
        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
            GLES20.glClearColor(0.5f,0.5f,0.5f, 1.0f);    //设置屏幕背景色 RGBA
            texRect=new Triangle(MySurfaceView.this);     //创建三角形对象
            GLES20.glEnable(GLES20.GL_DEPTH_TEST);        //打开深度检测
            initTexture();                                //初始化纹理
            GLES20.glDisable(GLES20.GL_CULL_FACE);        //关闭背面剪裁
      } }
      public void initTexture(){
           int[] textures = new int[1];                   //生成纹理 ID
           GLES20.glGenTextures(
                   1,                                     //产生的纹理 id 的数量
                   textures,                              //纹理 id 的数组
                   0                                      //偏移量
           );
           textureId=textures[0];                         //获取产生的纹理 id
           GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureId);//绑定纹理 id
           GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,   //设置 MIN 采样方式
               GLES20.GL_TEXTURE_MIN_FILTER,GLES20.GL_NEAREST);
           GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,   //设置 MAG 采样方式
               GLES20.GL_TEXTURE_MAG_FILTER,GLES20.GL_LINEAR);
           GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,   //设置 S 轴拉伸方式
               GLES20.GL_TEXTURE_WRAP_S,GLES20.GL_CLAMP_TO_EDGE);
           GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,   //设置 T 轴拉伸方式
               GLES20.GL_TEXTURE_WRAP_T,GLES20.GL_CLAMP_TO_EDGE);
         //通过输入流加载图片
         InputStream is = this.getResources().openRawResource( //建立指向纹理图的流
         R.drawable.wall);
         Bitmap bitmapTmp;
         try  {bitmapTmp = BitmapFactory.decodeStream(is); }    //从流中加载图片内容
         finally { try {is.close();} catch(IOException e) {e.printStackTrace();}}
```

```
52        GLUtils.texImage2D (                      //实际加载纹理进显存
53             GLES20.GL_TEXTURE_2D,                //纹理类型
54             0,                    //纹理的层次，0表示基本图像层，可以理解为直接贴图
55             bitmapTmp,                           //纹理图像
56             0                                    //纹理边框尺寸
57        );
58        bitmapTmp.recycle();       //纹理加载成功后释放内存中的纹理图
59        }}
```

- 第26行在onSurfaceCreated方法中调用initTexture方法初始化本案例中要用到的纹理。
- 第29-59行为初始化纹理的initTexture方法，其首先从系统获取分配的纹理id，然后设置此id对应纹理的采样与拉伸方式，接着通过流将纹理图加载进内存，最后将纹理图加载进显存并释放内存中的副本。最后的释放请读者务必记住，否则在纹理较多的项目中可能引起内存崩溃。
- 经过initTexture方法初始化后，后面再要使用纹理时就非常简单了，仅需要使用纹理id进行绑定即可，具体情况参看下一步Triangle类的开发。

提示

上述MySurfaceView中有很多部分的代码都省略了，这是因为大部分代码都与前面章节的代码很类似，需要的读者可以自行参考光盘中的源代码。

（2）接着开发的是场景中用于渲染三角形的类——Triangle，其代码框架如下。

代码位置：见随书光盘中源代码/第7章/Sample7_1/com/bn/Sample7_1目录下的Triangle.java。

```java
package com.bn.Sample7_1;                                   //包声明
import static com.bn.Sample7_1.ShaderUtil.createProgram; //相关类的引入
……//此处省略部分类引入的代码，请读者自行查看随书光盘的源代码
import android.opengl.GLES20;                               //相关类的引入
public class Triangle{                                      //纹理三角形
     int mProgram;                                          //自定义渲染管线着色器程序 id
    int muMVPMatrixHandle;                                  //总变换矩阵引用
    int maPositionHandle;                                   //顶点位置属性引用
    int maTexCoorHandle;                                    //顶点纹理坐标属性引用
        String mVertexShader;                               //顶点着色器代码脚本
        String mFragmentShader;                             //片元着色器代码脚本
         FloatBuffer   mVertexBuffer;                       //顶点坐标数据缓冲
         FloatBuffer   mTexCoorBuffer;                      //顶点纹理坐标数据缓冲
        int vCount=0;
        float xAngle=0;                                     //绕 x 轴旋转的角度
        float yAngle=0;                                     //绕 y 轴旋转的角度
        float zAngle=0;                                     //绕 z 轴旋转的角度
        public Triangle(MySurfaceView mv){
         initVertexData();                         //调用初始化顶点数据的 initVertexData 方法
         initShader(mv);                           //调用初始化着色器的 initShader 方法
        }
        public void initVertexData(){}             //初始化顶点属性数据的 initVertexData 方法
        public void initShader(MySurfaceView mv){}//自定义的初始化着色器的 initShader 方法
        public void drawSelf(int texId){
```

```
        GLES20.glUseProgram(mProgram);                    //指定使用某套着色器程序
        MatrixState.setInitStack();
          MatrixState.transtate(0, 0, 1);                 //设置沿 Z 轴正向位移 1
          MatrixState.rotate(yAngle, 0, 1, 0);            //设置绕 y 轴旋转指定角度
          MatrixState.rotate(zAngle, 0, 0, 1);            //设置绕 z 轴旋转指定角度
          MatrixState.rotate(xAngle, 1, 0, 0);            //设置绕 x 轴旋转指定角度
          //将最终变换矩阵传入渲染管线
          GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, false, MatrixState.getFinalMatrix(), 0);
          GLES20.glVertexAttribPointer(                   //将顶点位置数据传送进渲染管线
               maPositionHandle,
               3,
               GLES20.GL_FLOAT,                           //数据类型
               false,
             3*4,
             mVertexBuffer                                //数据缓冲
          );
          GLES20.glVertexAttribPointer(                   //将顶点纹理坐标数据传送进渲染管线
               maTexCoorHandle,
               2,
               GLES20.GL_FLOAT,                           //数据类型
               false,
             2*4,
             mTexCoorBuffer                               //数据缓冲
          );
          //允许用到的属性数据数组
          GLES20.glEnableVertexAttribArray(maPositionHandle);   //启用顶点位置数据
          GLES20.glEnableVertexAttribArray(maTexCoorHandle);    //启用顶点纹理坐标数据
          //绑定纹理
          GLES20.glActiveTexture(GLES20.GL_TEXTURE0);           //设置使用的纹理编号
          GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, texId);    //绑定指定的纹理 id
          //绘制纹理三角形
          GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount); //以三角形的方式填充
}}
```

- 第6-17行定义了一些成员变量，主要包括：自定义渲染管线着色器程序id、总变换矩阵的引用、顶点位置属性引用、顶点纹理坐标引用，三角形绕x 轴、y 轴、z 轴旋转的角度等。
- 第22-23行为初始化顶点属性数据的initVertexData方法以及初始化着色器的initShader方法，这里省略了具体代码，详细内容在后面给出。
- 第24-57行为绘制三角形的drawSelf方法，该方法中首先获得最终变换矩阵并传入渲染管线，然后将顶点坐标、顶点纹理坐标属性数据送入渲染管线，接着绑定纹理并绘制纹理三角形。

提示

请读者注意，用于绑定的纹理id必须是由前面initTexture方法初始化完成的，如上述代码的54行中的texId就是在MySurfaceView中由initTexture方法初始化，并在绘制时传递过来的。

（3）完成Triangle类代码框架的开发后，下面开发的是初始化着色器的initShader方法，具体 代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_1/com/bn/Sample7_1目录下的Triangle.java。

```
public void initShader(MySurfaceView mv){        //自定义初始化着色器的 initShader 方法
    //加载顶点着色器的脚本内容
   mVertexShader=ShaderUtil.loadFromAssetsFile("vertex.sh", mv.getResources());
   //加载片元着色器的脚本内容
   mFragmentShader=ShaderUtil.loadFromAssetsFile("frag.sh", mv.getResources());
   //基于顶点着色器与片元着色器创建程序
   mProgram = createProgram(mVertexShader, mFragmentShader);
   //获取程序中顶点位置属性引用
   maPositionHandle = GLES20.glGetAttribLocation(mProgram, "aPosition");
       //获取程序中顶点纹理坐标属性引用
       maTexCoorHandle= GLES20.glGetAttribLocation(mProgram, "aTexCoor");
       //获取程序中总变换矩阵引用
       muMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
  }
```

说明

initShader方法的主要作用为加载顶点着色器、片元着色器的脚本代码；基于顶点着色器与片元着色器创建着色程序；从着色程序中获取顶点坐标位置引用、顶点纹理属性引用等。

（4）完成了initShader方法的开发后，下面开发的是初始化顶点属性数据的initVertexData方法，具体代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_1/com/bn/Sample7_1目录下的Triangle.java。

```
public void initVertexData(){
    vCount=3;                                          //顶点的数量为3
    final float UNIT_SIZE=0.15f;
    float vertices[]=new float[]{                      //顶点坐标数据数组
     0*UNIT_SIZE,11*UNIT_SIZE,0,                       //上面顶点的坐标
     -11*UNIT_SIZE,- 11*UNIT_SIZE,0,                   //左下侧顶点的坐标
     11*UNIT_SIZE,- 11*UNIT_SIZE,0,                    //右下侧顶点的坐标
    };
    //创建顶点坐标数据缓冲
        ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
                                                       //创建恰当大小的缓冲区
        vbb.order(ByteOrder.nativeOrder());            //设置字节顺序为本地操作系统顺序
        mVertexBuffer = vbb.asFloatBuffer();           //转换为Float型缓冲
        mVertexBuffer.put(vertices);                   //向缓冲区写入顶点坐标数据
        mVertexBuffer.position(0);                     //设置缓冲区起始位置
        //顶点纹理坐标数据的初始化
        float texCoor[]=new float[]{                   //顶点纹理坐标数据数组
            0.5f,0,                                    //上面的点
            0,1,                                       //左下侧的点
            1,1                                        //右下侧的点
        };
```

```
21        //创建顶点纹理坐标数据缓冲
22        ByteBuffer cbb = ByteBuffer.allocateDirect(texCoor.length*4);
                                                   //创建恰当大小的缓冲区
23        cbb.order(ByteOrder.nativeOrder());      //设置字节顺序为本地操作系统顺序
24        mTexCoorBuffer = cbb.asFloatBuffer();    //转换为Float型缓冲
25        mTexCoorBuffer.put(texCoor);             //向缓冲区中写入数据
26        mTexCoorBuffer.position(0);              //设置缓冲区起始位置
27    }
```

- 第4-14行功能为初始化顶点坐标数据，并将数据存入数据缓冲中供绘制时使用。
- 第16-27行功能为初始化顶点纹理坐标数据，并将数据存入数据缓冲中供绘制时使用。

（5）完成了Java代码的开发后，下面将要开发的是顶点着色器，具体代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_1/assets目录下的vertex.sh文件。

```
uniform mat4 uMVPMatrix;                      //总变换矩阵
attribute vec3 aPosition;                     //顶点位置
attribute vec2 aTexCoor;                      //顶点纹理坐标
varying vec2 vTextureCoord;                   //用于传递给片元着色器的易变变量
void main() {
  gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点的位置
  vTextureCoord = aTexCoor;                   //将接收的纹理坐标传递给片元着色器
}
```

说明

第7行将被处理顶点的纹理坐标从属性变量aTexCoor赋值给了易变变量vTextureCoord,供渲染管线固定功能部分进行插值计算后传递给片元着色器使用。

（6）完成了顶点着色器的开发后，下面将要开发的是片元着色器，具体代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_1/assets目录下的frag.sh文件。

```
precision mediump float;                                  //指定默认浮点精度
varying vec2 vTextureCoord;                               //接收从顶点着色器过来的纹理坐标易变变量
uniform sampler2D sTexture;                               //纹理采样器，代表一幅纹理
void main() {
   gl_FragColor = texture2D(sTexture, vTextureCoord);     //进行纹理采样
}
```

- 此片元着色器的主要功能为根据接收的记录片元纹理坐标的易变变量中的纹理坐标，调用texture2D内建函数从采样器中进行纹理采样，得到此片元的颜色值。最后，将采样到的颜色值传给gl_FragColor内建变量，完成片元的着色。
- sampler2D类型的sTexture代表的就是前面步骤（2）中代码54行绑定的指定编号纹理的内容。

提示

该案例中还有几个与前面章节案例内容基本一样的类未作介绍，包括MyActivity类、ShaderUtil类、MatrixState类等，需要的读者可以自行参考光盘中的源代码。本章后继的案例也是如此，对于代码基本相同的类不再进行详细介绍。

有兴趣的读者还可以将本案例中三角形上面、左下、右下顶点的纹理坐标改为（0,0）、（0,1）、（1,1），再次运行本案例，其效果

如图7-4所示。

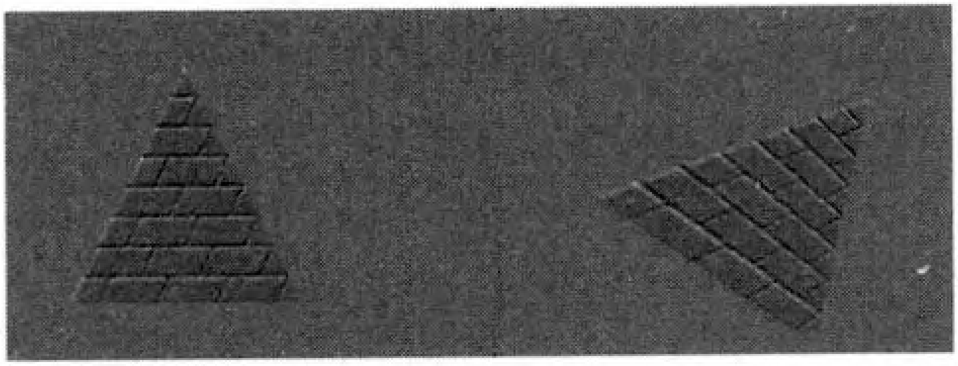


图7-4　修该纹理坐标后再次运行的效果图

> **说明**
>
> 图7-4中左侧为三角形原始姿态的情况，右侧是三角形旋转一定角度后的情况。

从图7-4中可以看出，纹理坐标更改后映射到三角形中的内容发生了变化，由于此次纹理坐标构成的三角形区域形状与三角形顶点坐标构成的三角形形状差异较大，所以纹理产生了变形。这也提醒我们在开发中如果不希望纹理映射后发生很大的变形，就需要尽量使构成图元的各顶点形成的几何形状与各顶点纹理坐标构成的几何形状相似。

7.2　纹理拉伸

上一小节中加载纹理的initTexture方法在对纹理进行设置时设置了S轴与T轴的拉伸方式，但没有对拉伸的问题进行详细的介绍。本小节将对纹理的两种不同拉伸方式进行详细的介绍，包括重复方式和截取方式。

7.2.1　两种拉伸方式概览

上一小节的案例中，无论是S轴还是T轴的纹理坐标都是在0.0～1.0中，这满足了大多数情况。但在特定的情况下，也可以设置大于1

的纹理坐标。当纹理坐标大于1以后，设置的拉伸方式就会起作用了。下面对两种拉伸方式进行单独的介绍，具体如下所列。

1．重复拉伸方式

若设置的拉伸方式为重复，当顶点纹理坐标大于1时则实际起作用的纹理坐标为纹理坐标的小数部分。也就是若纹理坐标为3.3，则起作用的纹理坐标为0.3，。这种情况下会产生重复的效果，如图7-5和图7-6所示。

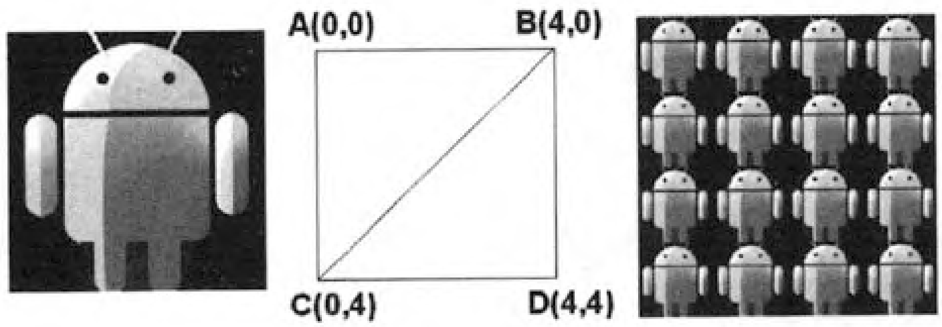


图7-5　重复纹理1

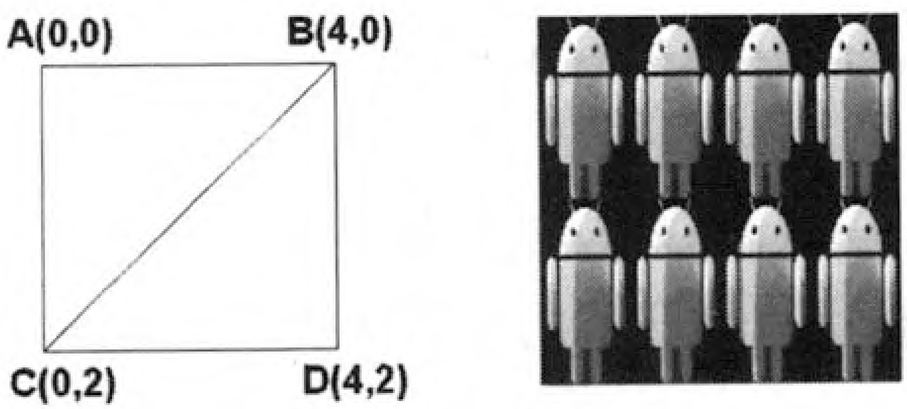


图7-6　重复纹理2

- 图7-5中左侧为原始纹理图，中间表示需要进行纹理映射的矩形的几何结构，其由4个顶点（两个三角形）组成，顶点纹理坐标分别为（0,0）、（4,0）、（0,4）、（4，4）。因此可以看出矩形中各片元的纹理坐标范围为S轴方向0～4，T轴方向0～4，应该在S、T两个轴

都产生重复4次的效果。右侧为纹理映射后的矩形，横向和纵向都产生了4次重复的效果。

- 图7-6中的情况与图7-5类似，只是4个顶点的纹理坐标改为了（0,0）、（4,0）、（0,2）、（4，2）。因此矩形中各片元的纹理坐标范围为S轴方向0～4，T轴方向0～2，应该在S轴方向重复4次，T轴方向重复2次。最后，右侧纹理映射后的情况也是如此。

纹理加载的过程中，设置纹理拉伸方式为重复方式的代码如下。

```
GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,            //设置S轴的拉伸方式为重复
    GLES20.GL_TEXTURE_WRAP_S, GLES20.GL_REPEAT);
GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,            //设置T轴的拉伸方式为重复
    GLES20.GL_TEXTURE_WRAP_T,GLES20.GL_REPEAT);
```

> **提示**
>
> 请读者注意开发中S与T两个轴的拉伸方式是独立设置的，但在一般情况下两个轴都会设置同样的选项。

重复拉伸方式在很多大场景地形的纹理贴图中很有作用，如将大块地面重复铺满草皮纹理、将大片水面重复铺满水波纹理等。如果没有重复拉伸方式，则开发人员只能将大块面积切割为一块一块的小面积矩形，对每一块矩形单独设置0.0～1.0内的纹理坐标。

这样开发不但烦琐，而且大大增加了顶点的数量，程序运行时的效率也会受到很大的影响。因此开发中要注意重复拉伸方式的灵活运用，可以使用重复拉伸方式时就不要去无谓地增加顶点数量了。

2. 截取拉伸方式

截取方式中当纹理坐标的值大于1时都看作1，因此会产生边缘被拉伸的效果，具体情况如图7-7所示。

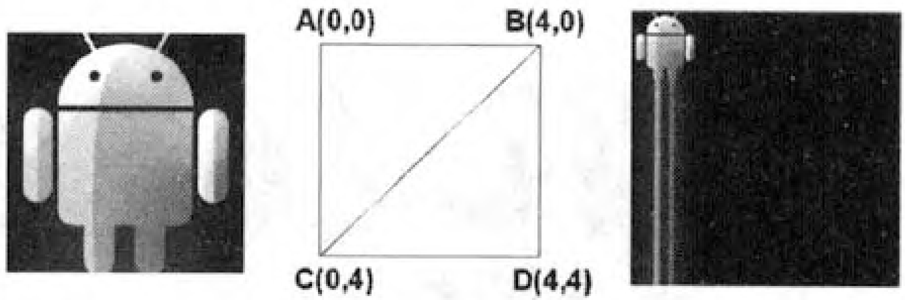


图7-7　截取纹理

从图7-7中可以看出，需要纹理映射的矩形中4个顶点纹理坐标分别为（0,0）、（4，0）、（0,4）、（4,4），因此矩形中的各片元纹理坐标范围为S轴方向0～4，T轴方向0～4。由于在此种拉伸方式下，大于1的纹理坐标都看作1，因此产生了纹理横向和纵向边缘被拉伸的效果。纹理加载的过程中，设置纹理拉伸方式为截取方式的代码如下。

```
GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,              //设置S轴的拉伸方式为截取
  GLES20.GL_TEXTURE_WRAP_S,GLES20.GL_CLAMP_TO_EDGE);
GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,              //设置T轴的拉伸方式为截取
  GLES20.GL_TEXTURE_WRAP_T,GLES20.GL_CLAMP_TO_EDGE);
```

提示

纹理拉伸在S与T轴方向上是独立的，可以在S方向上使用重复（GL_REPEAT）在T方向上使用截取（GL_CLAMP_TO_EDGE），反之亦然，但此种情况并不多见。另外，在实际开发中纹理坐标大于1并采用截取方式的情况并不多见，往往仅用于这种需要边缘拉伸的特殊效果时。

7.2.2　不同拉伸方式的案例

介绍完纹理拉伸的基本知识后，下面将通过一个简单的案例说明如何在开发中使用不同的纹理拉伸方式。此案例的功能为用不同的拉伸方式、不同的 行纹理映射，其运行效果如图7-8～图7-13所示。

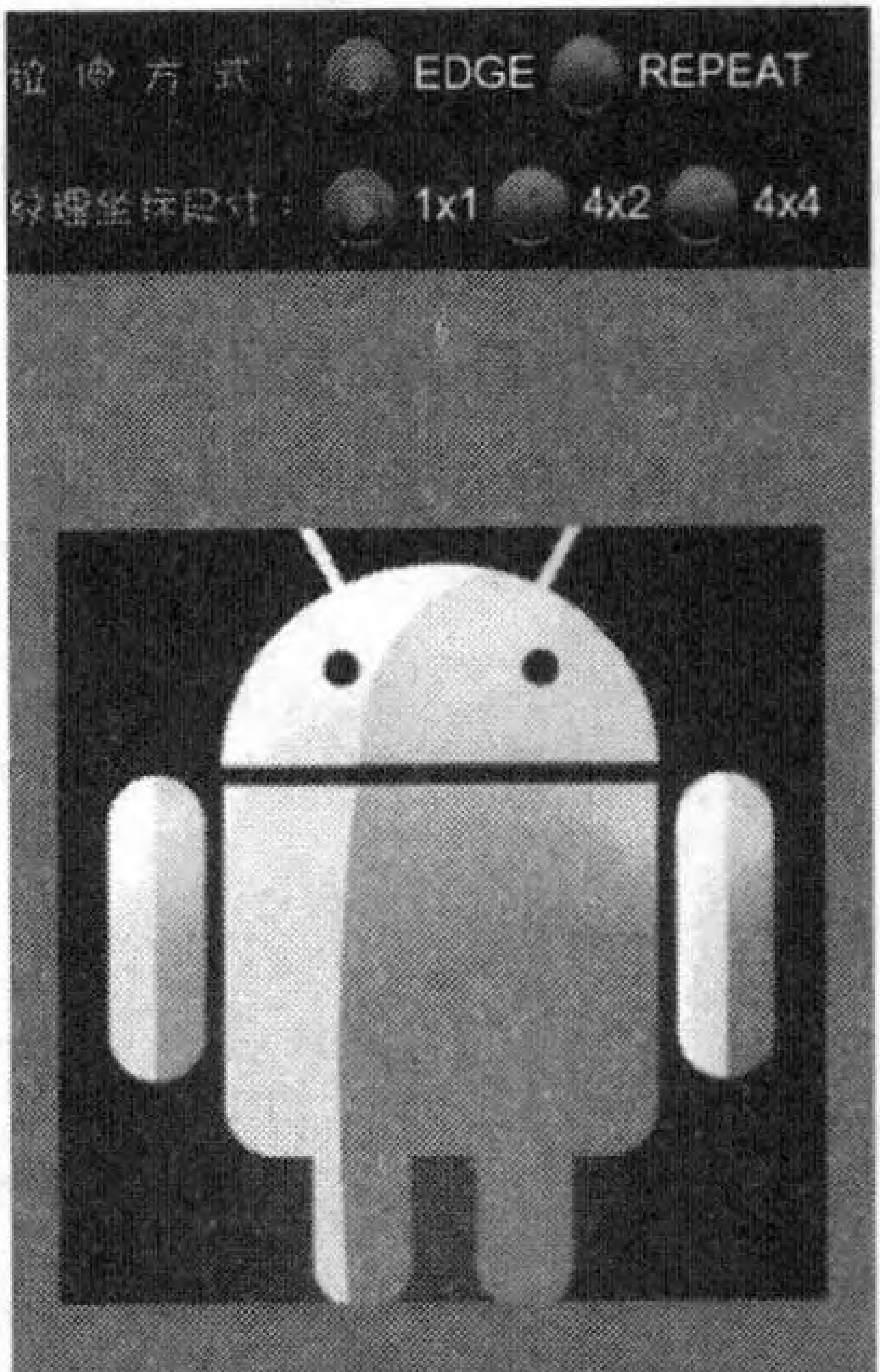

拉伸方式：
EDGE
REPEAT
纹理坐标尺寸：
1x1
4x2
4x4

图7-8　截取纹理1

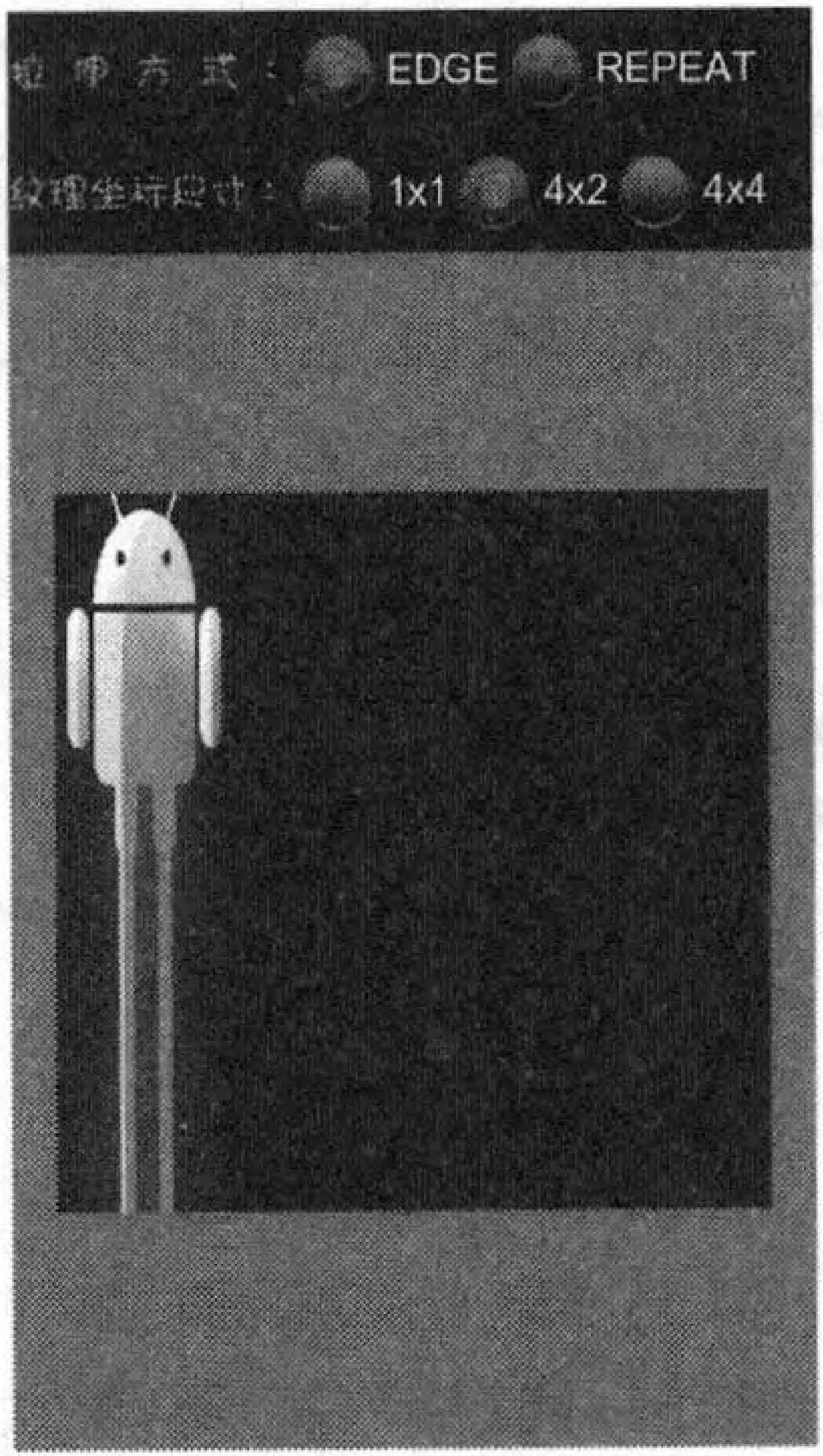

EDGE
REPEAT
1x1
4x2
4x4

图7-9　截取纹理2

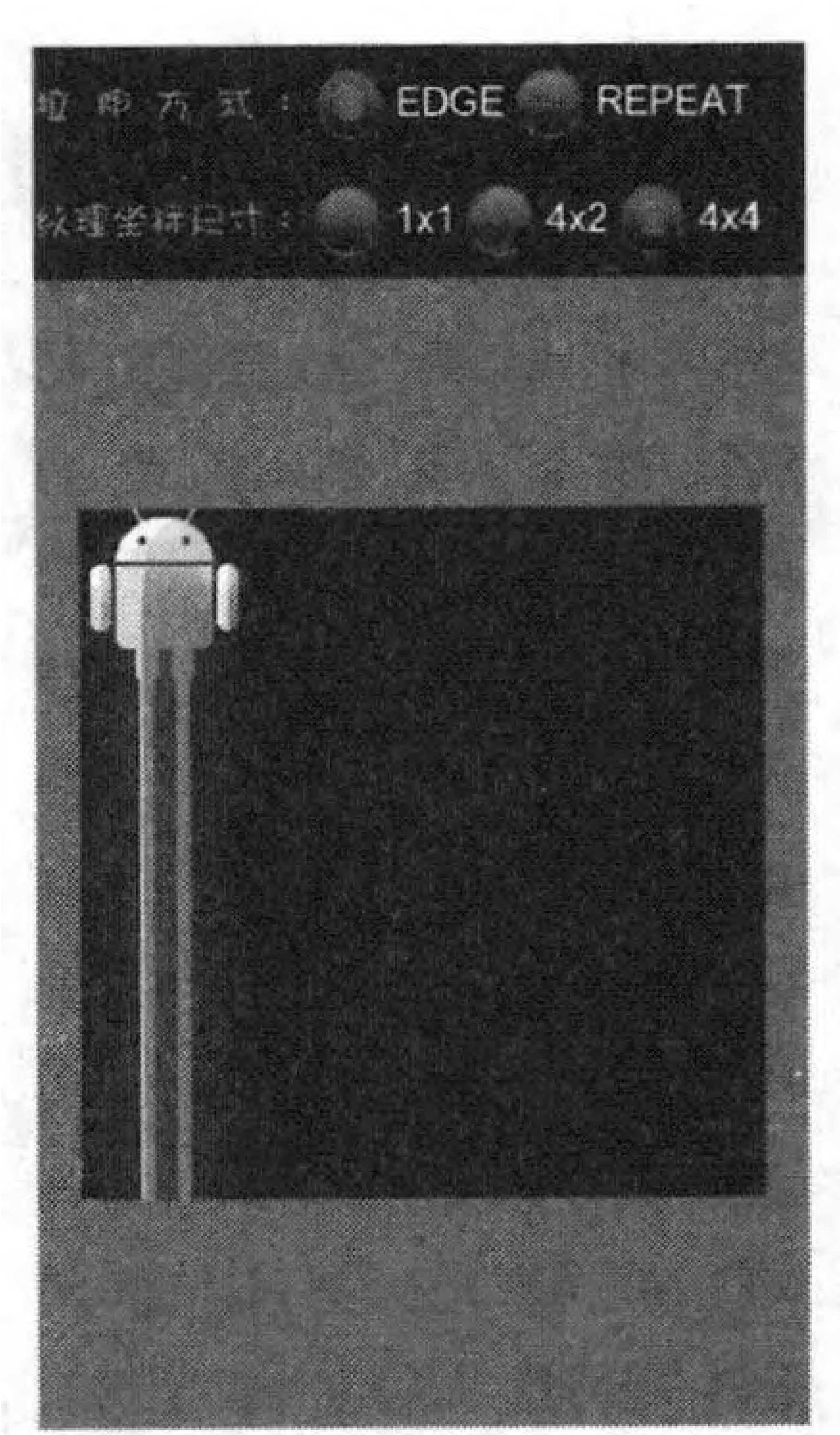


图7-10　截取纹理3

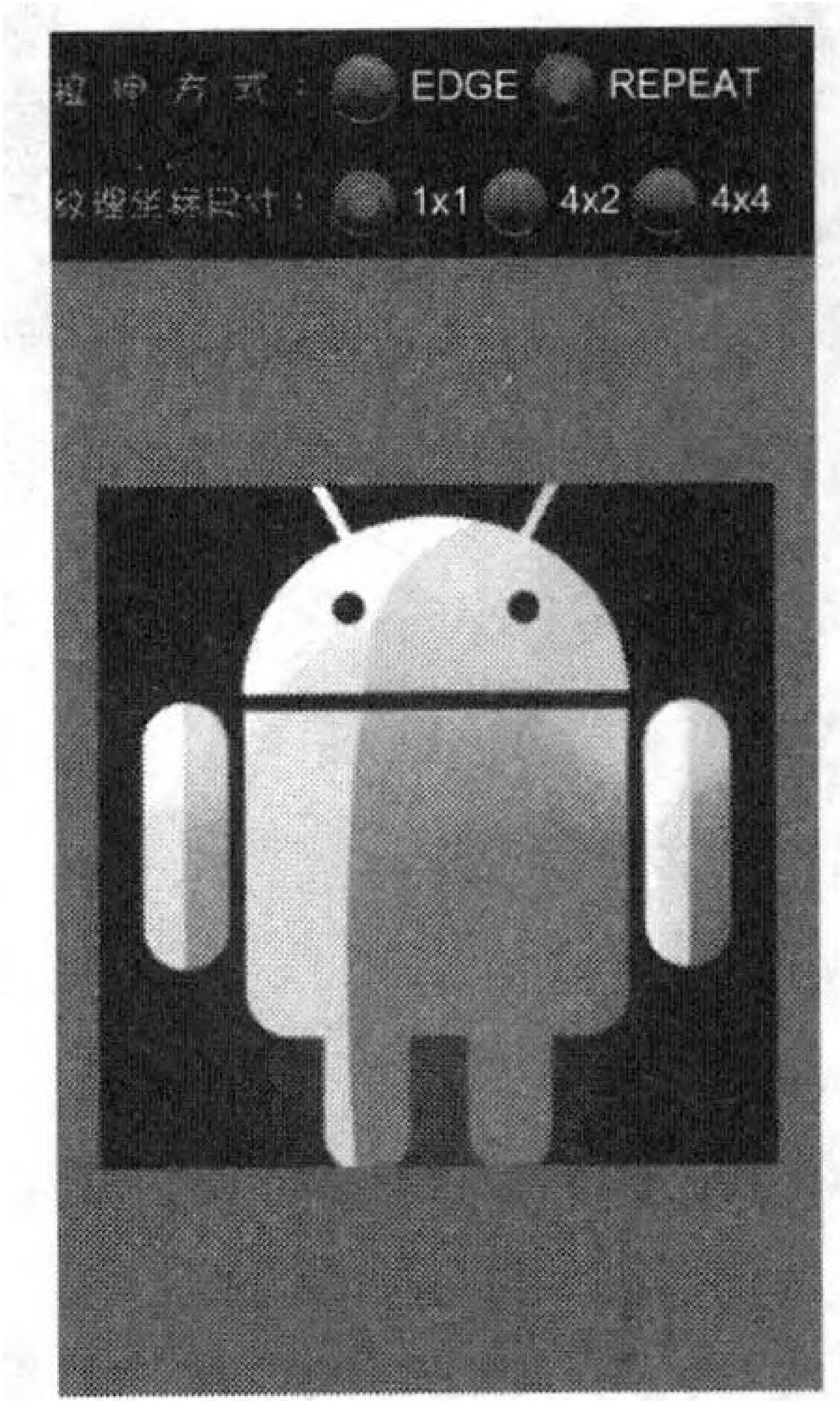


图7-11　重复纹理1

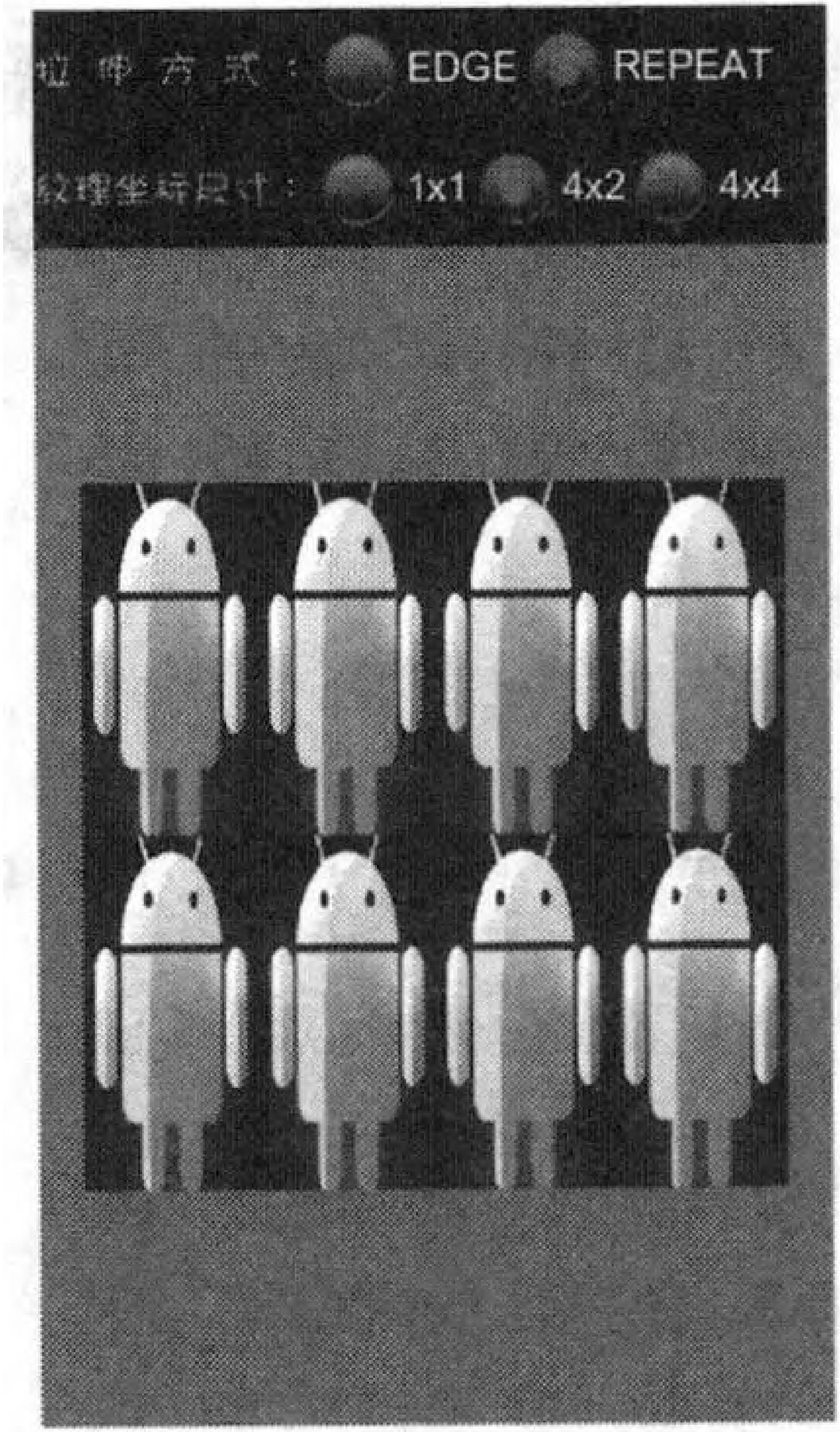

拉伸方式：
EDGE
REPEAT
纹理坐标尺寸：
1x1
4x2
4x4

图7-12　重复纹理2

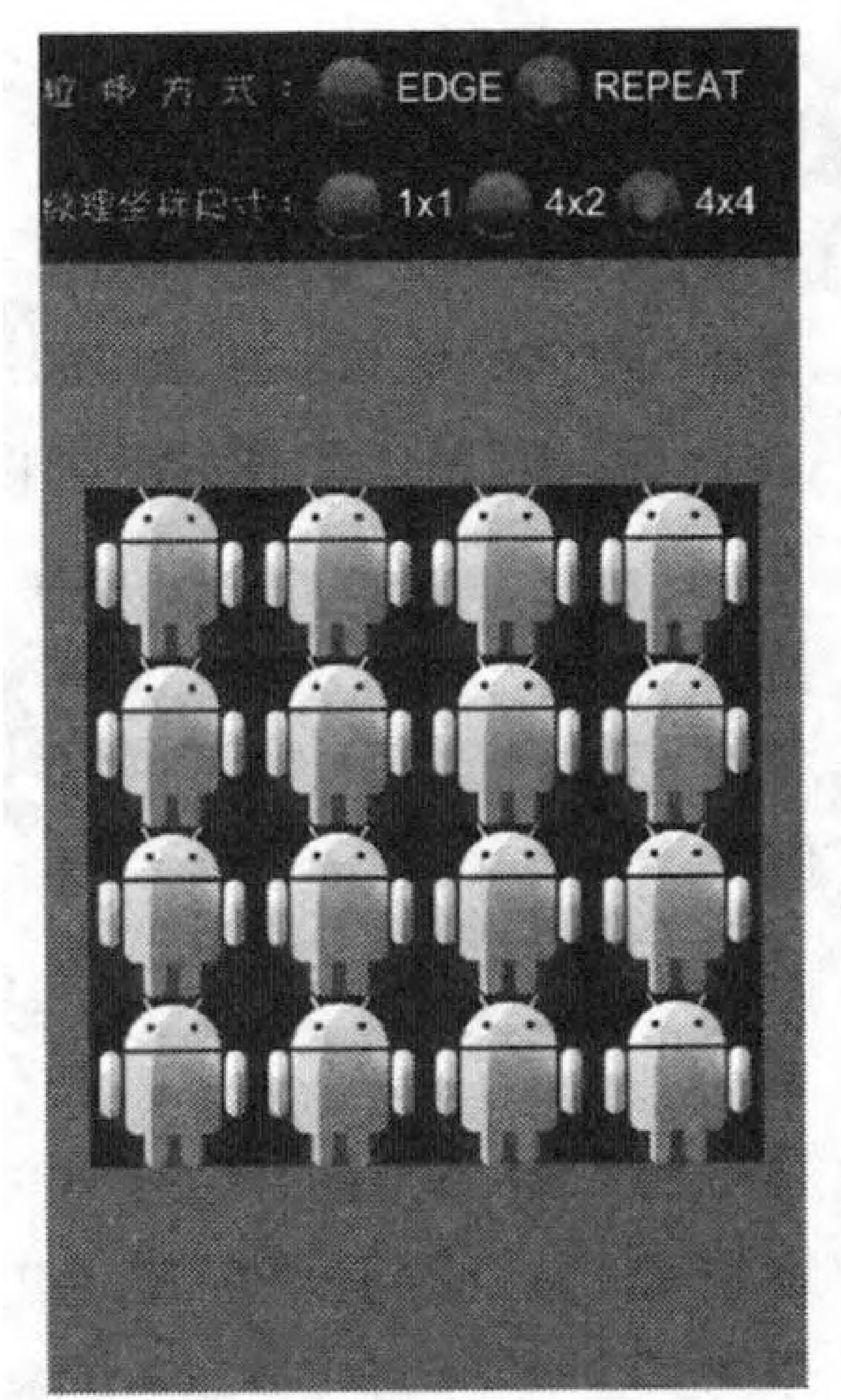


图7-13　重复纹理3

- 图7-8、图7-9、图7-10 3幅是程序在截取拉伸方式下的抓图，从左至右矩形的纹理坐标最大值分别为（1，1）、（4，2）、（4，4）。从3幅图中可以看出，在截取纹理拉伸方式下，纹理坐标中所有大于1.0的值均被设置为1.0使用，因此纹理边缘被拉伸。
- 图7-11、图7-12、图7-13 3幅是程序在重复拉伸方式下的抓图，从左至右矩形的纹理坐标最大值分别为（1，1）、（4，2）、（4，4）。从3幅图中可以看出，在重复纹理拉伸方式下，当纹理坐标大于1时，仅取其小数部分使用，因此产生了重复的效果。

前面已经介绍了本案例的运行效果，接下来将介绍本案例的具体开发步骤。

（1）首先开发的是该案例的Activity类，其中主要是进行了界面的布局及为“拉伸方式”与“纹理坐标尺寸”两组单选按钮添加了监听器。这部分界面控件相关的内容不是本书讨论的重点，需要的读者请自行参考随书光盘中的源代码。

（2）接着开发的是该案例中用来绘制整个场景的MySurfaceView类，该类中的大部分代码与本章第一个案例基本相同，主要的不同在于初始化纹理的initTexture方法，其代码如下。



代码位置：见随书光盘中源代码/第7/Sample7_2/com/bn/Sample7_2目录下的MySurfaceView.java。

```java
public int initTexture(boolean isRepeat){                       //初始化纹理的方法
    int[] textures = new int[1];                                //用于记录生成的纹理 id
    GLES20.glGenTextures(
            1,                                                  //产生的纹理 id 的数量
            textures,                                           //纹理 id 的数组
            0                                                   //偏移量
    );
    int textureId=textures[0];                                  //纹理 id
    GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureId);//绑定纹理
       GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
             GLES20.GL_TEXTURE_MIN_FILTER,GLES20.GL_NEAREST);//设定 MIN 的采样方式
       GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
             GLES20.GL_TEXTURE_MAG_FILTER,GLES20.GL_LINEAR); //设定 MAG 的采样方式
      if(isRepeat){
       GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,            //S 轴重复拉伸方式
             GLES20.GL_TEXTURE_WRAP_S,GLES20.GL_REPEAT);
          GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,         //T 轴重复拉伸方式
                GLES20.GL_TEXTURE_WRAP_T,GLES20.GL_REPEAT);
      }else{
       GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,            //S 轴截取拉伸方式
             GLES20.GL_TEXTURE_WRAP_S,GLES20.GL_CLAMP_TO_EDGE);
          GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,         //T 轴截取拉伸方式
                GLES20.GL_TEXTURE_WRAP_T,GLES20.GL_CLAMP_TO_EDGE);
      }
      //通过输入流加载图片
      InputStream is = this.getResources().openRawResource(R.drawable.tree);//打开输入流
      Bitmap bitmapTmp;
      try {
       bitmapTmp = BitmapFactory.decodeStream(is);          //从输入流中加载纹理图
      }finally { try {is.close();} catch(IOException e) {e.printStackTrace();}

      }
      GLUtils.texImage2D(              //实际加载纹理
          GLES20.GL_TEXTURE_2D,        //纹理类型
          0,                           //纹理的层次，0 表示基本图像层，可以理解为直接贴图
          bitmapTmp,                   //纹理图像
          0                            //纹理边框尺寸
      );
      bitmapTmp.recycle();             //纹理加载成功后释放图片
      return textureId;
    }
```

• 第14-24行为根据isRepeat参数的值设置不同的纹理拉伸方式，若其为true则设置为重复拉伸方式，否则设置为截取拉伸方式。

• 第25-40行为通过输入流将纹理图加载进内存，然后将纹理图加载进显存并释放内存中的复制。

提示

对于该类中没有详细介绍的部分，读者可自行查看随书光盘中的源代码。

（3）完成了MySurfaceView类的开发后，下面将要开发的是纹理矩形类TextureRect。首先开发的是该类的代码框架，具体代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_2/com/bn/Sample7_2目录下的TextureRect.java。

```java
package com.bn.Sample7_2;                                        //包声明
import static com.bn.Sample7_2.ShaderUtil.createProgram;         //相关类的引入
......//此处省略部分相关类的引入代码，请读者自行查看随书光盘的源代码
import android.opengl.Matrix;                                    //相关类的引入
public class TextureRect {                                       //纹理矩形
    int mProgram;                                        //自定义渲染管线着色器程序 id
    int muMVPMatrixHandle;                               //总变换矩阵引用
    int maPositionHandle;                                //顶点位置属性引用
    int maTexCoorHandle;                                 //顶点纹理坐标属性引用
        String mVertexShader;                            //顶点着色器代码脚本
        String mFragmentShader;                          //片元着色器代码脚本
        static float[] mMMatrix = new float[16];  //具体物体的 3D 变换矩阵，包括旋转、平移、缩放
         FloatBuffer   mVertexBuffer;                    //顶点坐标数据缓冲
         FloatBuffer   mTexCoorBuffer;                   //顶点纹理坐标数据缓冲
        int vCount=0;
        float xAngle=0;                                  //绕 x 轴旋转的角度
        float yAngle=0;                                  //绕 y 轴旋转的角度
        float zAngle=0;                                  //绕 z 轴旋转的角度
        float sRange;                                    //s 纹理坐标范围
        float tRange;                                    //t 纹理坐标范围
        public TextureRect(MySurfaceView mv,float sRange,float tRange){
         this.sRange=sRange;                             //接收 s 纹理坐标范围
         this.tRange=tRange;                             //接收 t 纹理坐标范围
         initVertexData();          //调用初始化顶点数据的 initVertexData 方法
         initShader(mv);            //调用初始化着色器的 initShader 方法
        }
        public void initVertexData(){             //初始化顶点数据的 initVertexData 方法
         //顶点坐标数据的初始化
           vCount=6;
           final float UNIT_SIZE=0.3f;            //设置单位长度
           float vertices[]=new float[]{          //顶点坐标数组
             -4*UNIT_SIZE,4*UNIT_SIZE,0,          //顶点坐标
             -4*UNIT_SIZE,-4*UNIT_SIZE,0,         //顶点坐标
              4*UNIT_SIZE,-4*UNIT_SIZE,0,         //顶点坐标
              4*UNIT_SIZE,-4*UNIT_SIZE,0,         //顶点坐标
              4*UNIT_SIZE,4*UNIT_SIZE,0,          //顶点坐标
             -4*UNIT_SIZE,4*UNIT_SIZE,0           //顶点坐标
```

```
38          };
39          //创建顶点坐标数据缓冲
40          ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
41          vbb.order(ByteOrder.nativeOrder());          //设置字节顺序为本地操作系统顺序
42          mVertexBuffer = vbb.asFloatBuffer();         //转换为Float型缓冲
43          mVertexBuffer.put(vertices);                 //向缓冲区中放入写入数据
44          mVertexBuffer.position(0);                   //设置缓冲区起始位置
45          //顶点纹理坐标数据的初始化
46          float texCoor[]=new float[]{                 //顶点纹理数组
47                  0,0, 0,tRange, sRange,tRange,
48                  sRange,tRange, sRange,0, 0,0         //纹理坐标
49           };
50          //创建顶点纹理坐标数据缓冲
51          ByteBuffer cbb = ByteBuffer.allocateDirect(texCoor.length*4);
52          cbb.order(ByteOrder.nativeOrder());          //设置字节顺序为本地操作系统顺序
53          mTexCoorBuffer = cbb.asFloatBuffer();        //转换为Float型缓冲
54          mTexCoorBuffer.put(texCoor);                 //向缓冲区中写入数据
55          mTexCoorBuffer.position(0);                  //设置缓冲区起始位置
56      }
57      public void initShader(MySurfaceView mv){/*代码省略*/}  //初始化着色器的initShader方法
58      public void drawSelf(int texId){ /*代码省略*/}     //绘制纹理矩形的方法
59  }
```

- 第6-20行定义了一些需要使用的成员变量，主要包括：自定义渲染管线程序id、总变换矩阵引用、顶点位置属性引用、顶点纹理坐标属性引用等。
- 第27-56行为对顶点数据进行初始化的initVertexData方法，其代码与本章中第一个三角形案例的基本相同。最大的区别是纹理坐标是通过接收的参数来设置的，不再是固定的，灵活性有所增强，满足了本案例中矩形顶点纹理坐标变化的需要。

- 对于初始化着色器的initShader方法、绘制纹理矩形的drawSelf方法，由于与前面的案例没有太大区别，套路完全相同，因此不再赘述。

> **提示**
>
> 该案例中自然也用到了顶点着色器与片元着色器，但由于本案例中的顶点着色器与片元着色器与本章第一个案例中的完全相同，因此这里也不再赘述，需要的读者可自行查看随书光盘中的源代码。

7.3　纹理采样

上一节介绍了纹理的两种不同拉伸方式，本节主要介绍两种不同的纹理采样方式：最近点采样与线性采样。通过这一节的学习读者应该能够掌握两种不同纹理采样方式的原理，并能够根据具体情况选用恰当的纹理采样方式。

7.3.1　纹理采样概述

前面的章节也简单提到过，所谓纹理采样就是根据片元的纹理坐标到纹理图中提取对应位置颜 色的过程。但由于被渲染图元中的片元数量与其对应纹理区域中像素的数量并不一定相同，也就是说图元中的片元与纹理图中的像素并不总是一一对应的。

例如，将较小的纹理图映射到较大的图元或将较大的纹理图映射到较小的图元时这种情况就会产生。因此通过纹理坐标在纹理图中并不一定能找到与之完全对应的像素，这时候就需要采用一些策略使得纹理采样可以顺利进行下去。通常采用的策略有最近点采样、线性采样两种，下面的小节将一一进行详细介绍。

7.3.2　最近点采样

最近点采样是最简单的一种采样算法，其速度在各种采样算法中也是最快的，本小节将分基本原理与效果特点两个方面对其进行介绍。

1．基本原理

最近点采样算法的基本原理如图7-14所示。

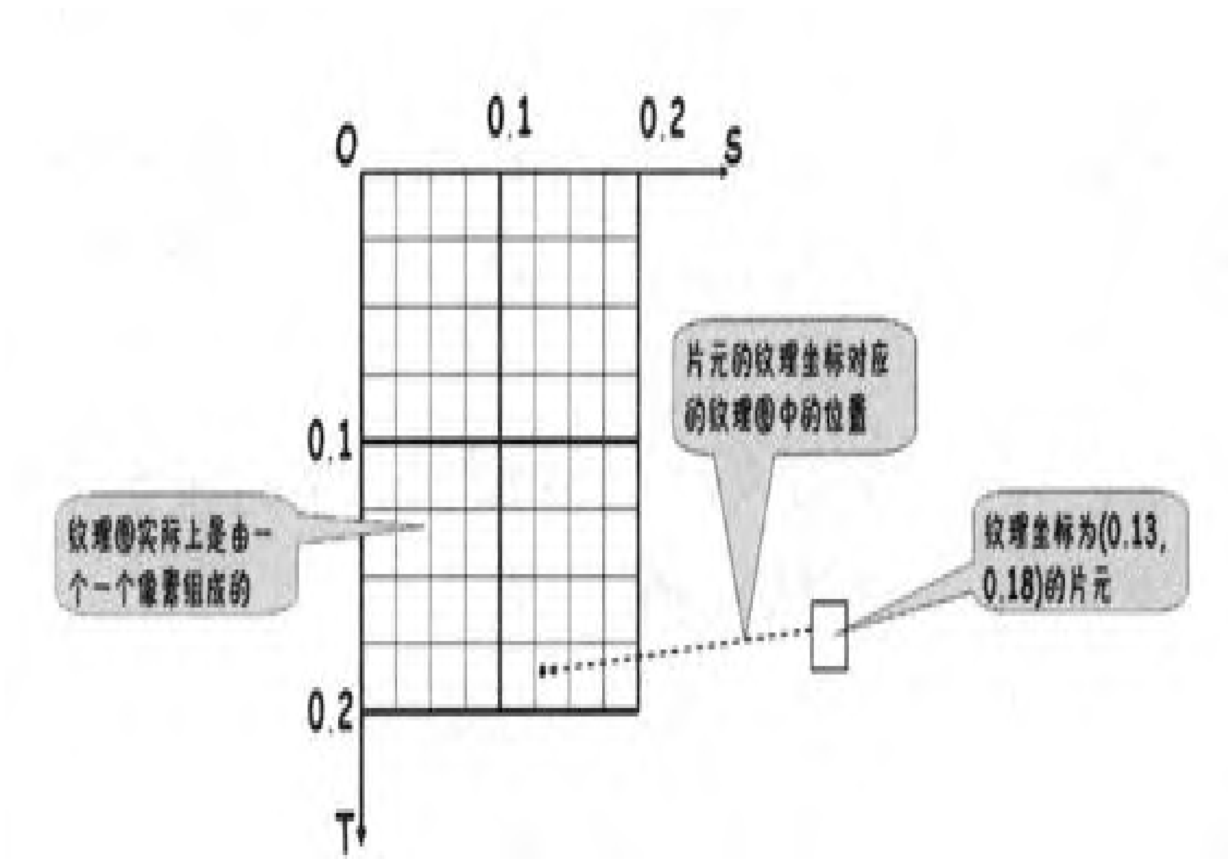


图7-14　最近点采样的原理图

从图7-14中可以看出两点。

- 纹理图的横边和竖边纹理坐标的范围都是0～1，而纹理图本身是由一个一个离散的像素组成的。若将每个像素看成一个小方块，则每个像素都占一定的纹理坐标范围。如图7-14中纹理图最左上侧的像素纹理坐标的范围为：S方向0.0～0.025，T方向0.0～0.025。
- 根据片元的纹理坐标可以很容易地计算出片元对应的纹理坐标点位于纹理图中的哪个像素（小方格）中，最近点采样就直接取此像素的颜色值为采样值。

2．效果特点

从前面原理的介绍中可以看出，最近点采样很简单，计算量也小。但最近点采样也有一个明显的缺点，那就是若把较小的纹理图映射到较大的图元上时容易产生很明显的锯齿，如图7-15所示。

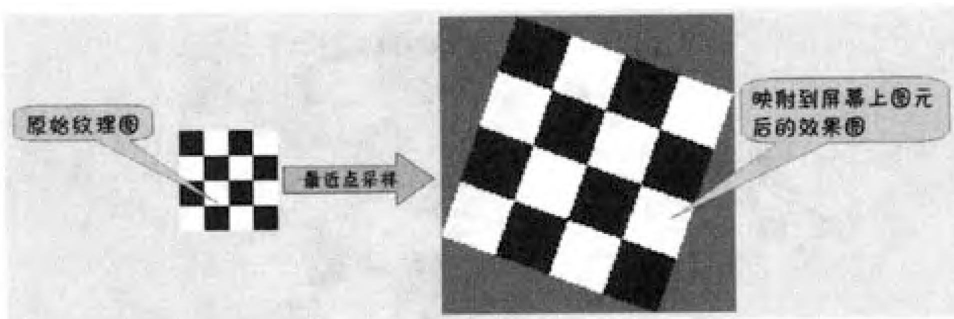


图7-15　最近点采样的特点图

需要注意的是将较大的纹理图映射到较小的图元时，也会有锯齿产生，但由于图元整体较小，视觉上就不那么明显了。

加载纹理时，设置采用最近点采样方式的代码如下。

```
GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,     //设置MIN时为最近点采样
            GLES20. GL_TEXTURE_MIN_FILTER, GLES20.GL_NEAREST);
GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,     //设置MAG时为最近点采样
            GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_NEAREST);
```

提示

关于上述代码中GL_TEXTURE_MIN_FILTER和GLES20.GL_TEXTURE_MAGFILTER的含义将在后继的小节中给出，这里读者简单了解即可。

7.3.3　线性纹理采样

从上一小节的介绍中可以看出，在某些情况下，最近点采样不能满足高视觉效果的要求，这时可以选用更复杂一些的线性纹理采样算法。本小节将对线性纹理采样算法进行介绍，主要包括基本原理及效果特点两方面。

1．基本原理

线性采样算法的原理如图7-16所示。

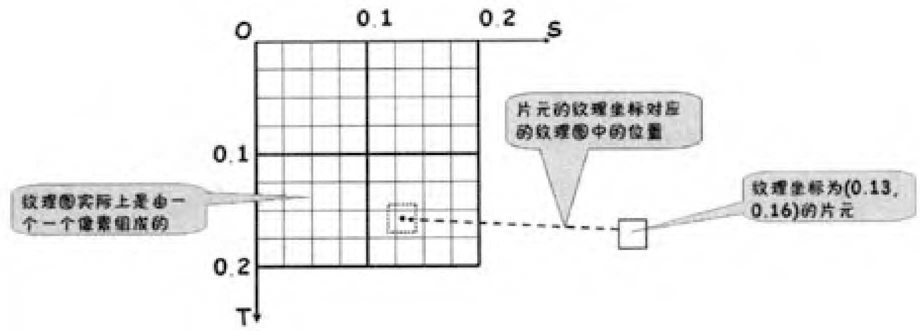


图7-16　线性纹理采样的原理图

线性采样时的结果颜色并不一定仅来自于纹理图中的一个像素，其在采样时会考虑到片元对应的纹理坐标点附近的几个像素。如图7-16所示，右侧片元的纹理坐标对应的纹理点在纹理图中的小黑点位置，此时可以认为纹理点位于采样范围的中央。因此采样范围涉及了4个像素，但由于纹理点并没有位于4个像素的交叉点上，从而4个像素的颜色在结果中所占的比例也不尽相同。

一般是根据涉及的像素在采样范围内的面积比例加权计算出最终的采样结果，但具体采样时使用的采样范围可能因厂商的不同而有所不同，上述示意图中的采样范围只是对原理的说明。

2．效果特点

从前面线性采样基本原理的介绍中可以推导出，由于采样时对采样范围内的多个像素进行了加 权平均，因此在将较小的纹理图映射到较大的图元上时，不再会有锯齿的现象，而是平滑过渡的，如图7-17所示。

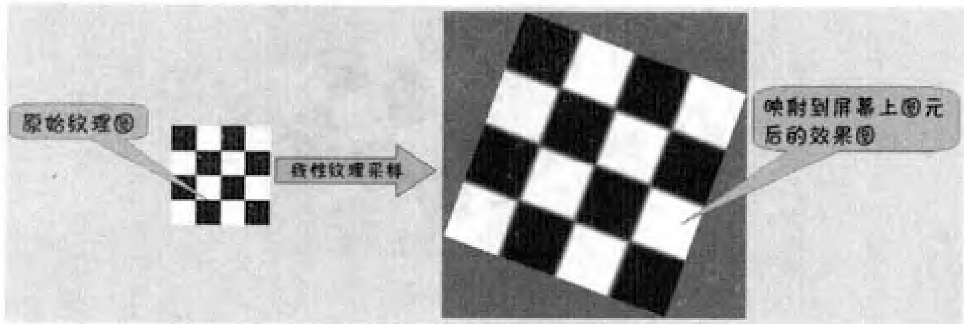


图7-17　线性纹理采样的特点图

> **提示**
>
> 平滑过渡解决了锯齿的问题，但有时线条边缘会很模糊，因此实际开发中采用哪种采样策略，需要根据具体的需求来确定。

加载纹理时，设置采用线性采样方式的代码如下。

```
GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,        //设置MIN时为线性采样
            GLES20.GL_TEXTURE_MIN_FILTER, GLES20.GL_LINEAR);
GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,        //设置MAG时为线性采样
            GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_LINEAR);
```

> **提示**
>
> 关于上述代码中GL_TEXTURE_MIN_FILTER和GLES20.GL_TEXTURE_MAG_FILTER的含义将在后继的小节中给出，这里读者简单了解即可。

7.3.4　MIN与MAG采样

前两个小节介绍了两种不同的纹理采样方式，读者应该已经注意到，无论采用哪种采样方式，都需要对

MIN（GL_TEXTURE_MIN_FILTER）与MAG（GL_TEXTURE_MAG_FILTER）两种情况分别进行设置。本小节将介绍MIN与MAG两种纹理采样情况的具体含义，其原理分别如图7-18和图7-19所示。

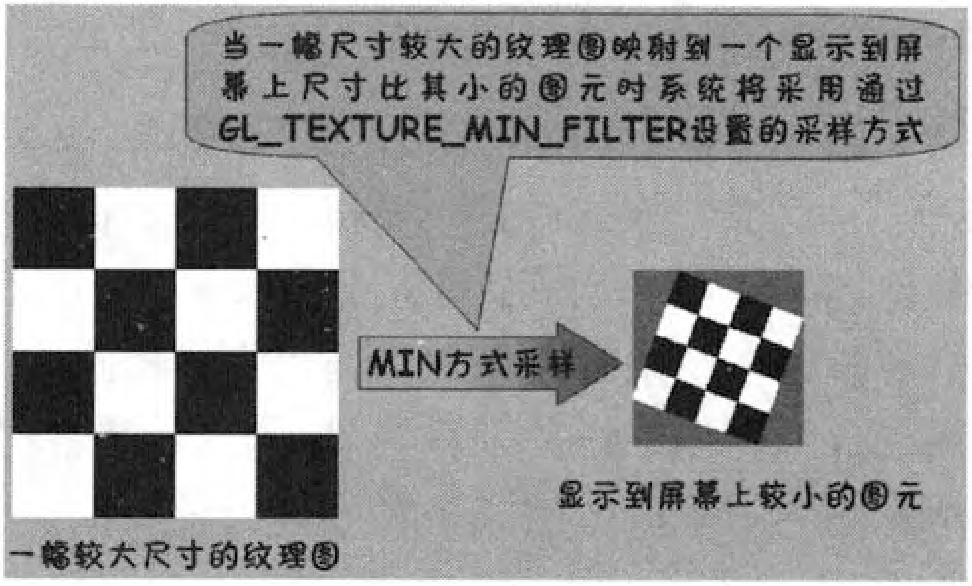


图7-18　MIN采样

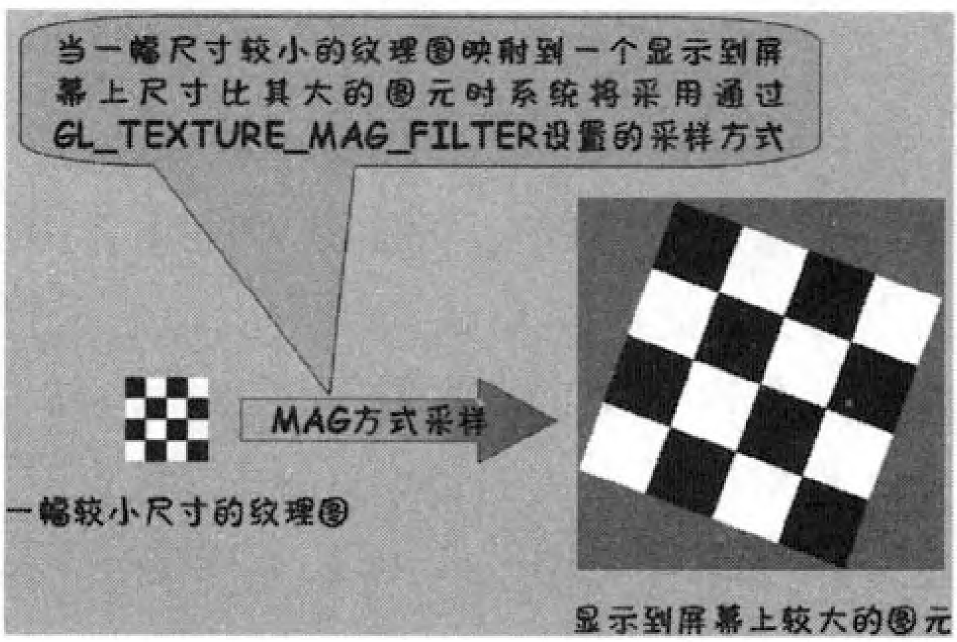


图7-19　MAG采样

从图7-18与图7-19中可以看出，当纹理图比需要映射的图元尺寸大时系统采用MIN对应的纹 理采样算法设置，而当纹理图比需要映射的图元尺寸小时系统采用MAG对应的纹理采样算法设置。

上述是通俗的说法，更准确的定义应该是：当纹理图中的一个像素对应到待映射图元上的多个片元时，采用MAG采样；反之则采用MIN采样。

> **提示**
>
> 由于最近点采样计算速度快，在MIN情况下一般锯齿也不明显，综合效益高;而MAG方式下若采用最近点采样则锯齿会很明显，严重影响视觉效果。因此实际开发中一般情况下往往采用将MIN情况设置为最近点采样，将MAG情况设置为线性采样的组合。

7.3.5　不同纹理采样方式的案例

前面几个小节对纹理采样各方面的知识进行了介绍，本小节将给出一个综合运用了这些知识的案例，其运行效果如图7-20所示。

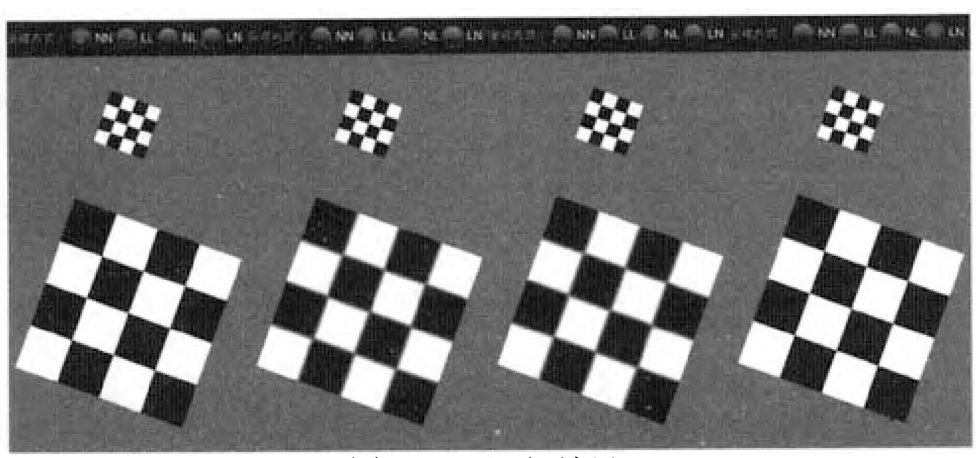


图7-20　运行效果

- 本案例中上方较小的纹理矩形是用尺寸比其大的纹理图映射的，用来演示MIN情况；下方较大的纹理矩形是用尺寸比其小的纹理图映射的，用来演示MAG情况。

- 最上面采样方式的4个单选按钮由左至右依次代表MIN和MAG情况的不同采样方式组合，具体情况为：最近点采样与最近点采样组合（NN）、线性采样与线性采样组合（LL）、最近点采样与与线性采样组合（NL）、线性采样与最近点采样组合（LN）。
- 最左侧的运行效果图为较大的与较小的纹理矩形均采用最近点采样方式的运行效果。
- 左起第二幅效果图为较大的与较小的纹理矩形均采用线性采样方式的运行效果。
- 左起第三幅效果图为较小的纹理矩形采用最近点采样方式，较大的纹理矩形采用线性采样方式的运行效果。
- 最右侧的效果图为较小的纹理矩形采用线性采样方式，较大的纹理矩形采用最近点采样方式的运行效果。

由于图片是缩小排版的，可能线条边缘的锯齿或模糊情况并不明显。下面将最近点采样的运行效果图部分进行放大，这样读者就可以很清楚地看到线条边缘的锯齿了，如图7-21所示。

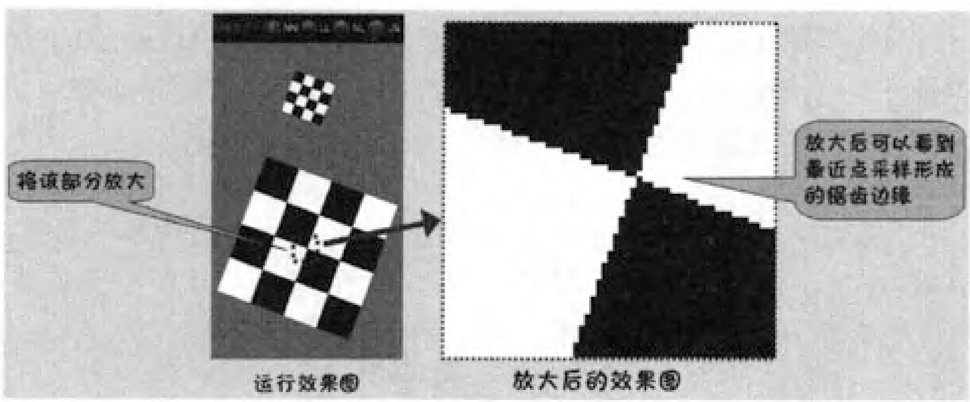


图7-21　采用最近点采样后运行效果图中部分放大后的效果

可以将线性采样的运行效果图也进行部分放大，这时可以很清楚地看到线条的边缘是平滑过渡的，并没有锯齿现象，如图7-22所示。

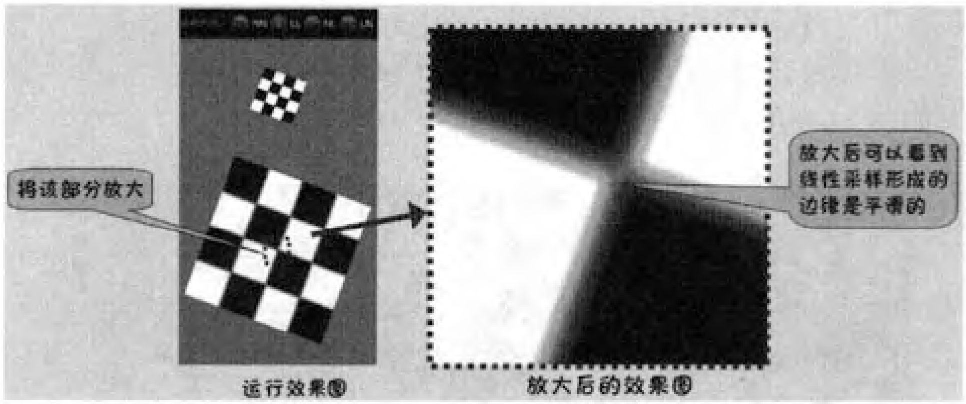


图7-22　采用线性采样后运行效果图中部分放大后的效果

提示

从上述的几幅图中可以看出，在MAG情况下，不同的采样方式有非常明显的区别；而在MIN情况下，两种采样方式区别不大。因此从提高执行效率的角度出发，在MIN情况下一般设置为最近点采样。另外，某些特殊情况可能不希望边缘模糊，此时MAG情况也会采用最近点采样。

介绍完了本案例的运行效果后，下面介绍本案例的具体开发步骤，具体如下所列。

（1）首先开发的是本案例的主控制类MyActivity，其代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_3/com/bn/Sample7_3目录下的MyActivity.java。

```java
package com.bn.Sample7_3;                                          //包声明
import android.app.Activity;                                       //相关类的引入
……//此处省略部分相关类的引入代码，读者可自行查看随书光盘的源代码
import android.widget.CompoundButton.OnCheckedChangeListener;//相关类的引入
public class MyActivity extends Activity {
    private MySurfaceView mGLSurfaceView;                         // MySurfaceView 类的引用
   @Override
   protected void onCreate(Bundle savedInstanceState) {          //重写 onCreate 方法
      super.onCreate(savedInstanceState);                         //调用父类的方法
          requestWindowFeature(Window.FEATURE_NO_TITLE);      //设置为无标题
          getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN
            WindowManager.LayoutParams.FLAG_FULLSCREEN);       //设置为全屏
            //设置为横屏模式
```

```
14            setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
15            setContentView(R.layout.main);                 //切换到主界面
16          mGLSurfaceView = new MySurfaceView(this);        //初始化GLSurfaceView
17          mGLSurfaceView.requestFocus();                   //获取焦点
18          mGLSurfaceView.setFocusableInTouchMode(true);    //设置为可触控
19          //将自定义的MySurfaceView添加到外层LinearLayout中
20          LinearLayout ll=(LinearLayout)findViewById(R.id.main_liner);
21          ll.addView(mGLSurfaceView);
22         RadioButton rb=(RadioButton)findViewById(R.id.Radi01); //获取RadioButton对象
23          rb.setOnCheckedChangeListener(               //为RadioButton添加监听器
24             new OnCheckedChangeListener(){
25                @Override                               //重写方法的标志
26                public void onCheckedChanged(CompoundButton buttonView,boolean
                  isChecked) {
27                    if(isChecked){                      //如果当选按钮被按下
28                        mGLSurfaceView.currenttexId32 = mGLSurfaceView.texId[0];
29                        mGLSurfaceView.currenttexId256 = mGLSurfaceView.texId[4];
30            }}});
31          ……//此处省略了其余3个RadioButton控件的监听事件
32       }
33       ……//此处省略了onResume与onPause方法的代码
34   }
```

• 第10-21行代码功能为设置为全屏模式、设置为无标题模式、设置为横屏模式、获取焦点、设置为可触控，并且将MySurfaceView类的对象添加到LinearLayout布局之中。

• 第22-31行获取了应用程序中4个单选按钮的引用，并为他们各自添加了监听器。这些单选按钮用来供用户单击以选择不同的纹理采样组合。

提示

上述代码中省略了一些与前面案例基本一致的部分，需要的读者请自行参考光盘中的源代码。

（2）完成了主控制类MyActivity的开发后，下面需要开发的是用于渲染整个场景的MySurfaceView类，其代码框架如下。

代码位置：见随书光盘中源代码/第7章/Sample7_3/com/bn/Sample7_3目录下的MySurfaceView.java。

```
package com.bn.Sample7_3;                                   //包声明
import java.io.IOException;                                 //相关类的引入
……//此处省略部分相关类的引入代码，读者可自行查看随书光盘的源代码
import android.graphics.BitmapFactory;                      //相关类的引入
class MySurfaceView extends GLSurfaceView {
    private SceneRenderer mRenderer;                        //场景渲染器引用
    int currenttexId32;                                     //大纹理矩形当前纹理的id
    int currenttexId256;                                    //小纹理矩形当前纹理的id
    int[] texId = new int[8];                               //存储系统分配的纹理id的数组
        public MySurfaceView(Context context) {/*代码省略，需要的读者参考随书光盘*/}
        private class SceneRenderer implements GLSurfaceView.Renderer {
        TextureRect texRect;                                //纹理矩形对象的引用
```

```java
        public void onDrawFrame(GL10 gl) {
            GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT |
            GLES20.GL_COLOR_BUFFER_BIT);                    //清除深度缓冲与颜色缓冲
            //绘制小纹理矩形
            MatrixState.pushMatrix();                       //保存当前的变换矩阵
            MatrixState.translate(0, 1.3f, 1);              //平移
            MatrixState.rotate(-20, 0, 0, 1);               //旋转
            MatrixState.scale(0.3f, 0.3f, 0.3f);            //缩放
            texRect.drawSelf(currenttexId256);              //绘制小纹理矩形
            MatrixState.popMatrix();                        //恢复当前的变换矩阵
            //绘制大纹理矩形
            MatrixState.pushMatrix();                       //保存当前的变换矩阵
            MatrixState.translate(0, -0.6f, 1);             //平移
            MatrixState.rotate(-20, 0, 0, 1);               //旋转
            texRect.drawSelf(currenttexId32);               //绘制大纹理矩形
            MatrixState.popMatrix();                        //恢复当前的变换矩阵
        }
        public void onSurfaceChanged(GL10 gl, int width, int height) {
          GLES20.glViewport(0, 0, width, height);          //设置视窗大小及位置
            float ratio = (float) width / height;           //计算GLSurfaceView的宽高比
            MatrixState.setProject(-ratio, ratio, -1,1,1,10);
                                                            //调用此方法计算产生透视投影矩阵
            MatrixState.setCamera(0,0,3,0f,0f,0f,0f,1.0f,0.0f);
                                                       //调用此方法产生摄像机9参数位置矩阵
            MatrixState.setInitStack();                     //初始化变换矩阵
        }
       public void onSurfaceCreated(GL10 gl, EGLConfig config) {/*具体代码后面给出*/}
      }
    public int initTexture(int drawableId,float sample1,float sample2){
    /*具体代码后面给出*/}
}
```

说明

上述代码中最主要的就是onDrawFrame与onSurfaceChanged两个方法，onDrawFrame方法的功能为在屏幕上的指定位置绘制一大一小两个纹理矩形，onSurfaceChanged方法主要是进行绘制前的一些准备工作，如设置视口、投影、摄像机等。

（3）完成了MySurfaceView类代码框架的开发后，下面需要开发的是每次场景创建时执行一次的onSurfaceCreated方法，其代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_3/com/bn/Sample7_3目录下的MySurfaceView.java。

```
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    GLES20.glClearColor(0.5f,0.5f,0.5f, 1.0f);         //设置屏幕背景色RGBA
    texRect = new TextureRect(MySurfaceView.this);    //创建纹理矩形对象
    GLES20.glEnable(GLES20.GL_DEPTH_TEST);             //打开深度检测
    //初始化32*32像素的4幅纹理
    texId[0] = initTexture(R.drawable.bw32,GLES20.GL_NEAREST,GLES20.GL_NEAREST);
```

```java
    texId[1] = initTexture(R.drawable.bw32,GLES20.GL_LINEAR,GLES20.GL_LINEAR);
    texId[2] = initTexture(R.drawable.bw32,GLES20.GL_NEAREST,GLES20.GL_LINEAR);
    texId[3] = initTexture(R.drawable.bw32,GLES20.GL_LINEAR,GLES20.GL_NEAREST);
        //初始化256*256像素的4幅纹理
        texId[4] = initTexture(R.drawable.bw256,GLES20.GL_NEAREST,GLES20.GL_NEAREST);
        texId[5] = initTexture(R.drawable.bw256,GLES20.GL_LINEAR,GLES20.GL_LINEAR);
        texId[6] = initTexture(R.drawable.bw256,GLES20.GL_NEAREST,GLES20.GL_LINEAR);
        texId[7] = initTexture(R.drawable.bw256,GLES20.GL_LINEAR,GLES20.GL_NEAREST);
        currenttexId32 = texId[0];                        //设置当前纹理
        currenttexId256 = texId[4];                       //设置当前纹理
        GLES20.glDisable(GLES20.GL_CULL_FACE);            //关闭背面剪裁
    }
```

说明

上述代码首先创建了用于绘制纹理矩形的TextureRect类对象，然后采用不同的采样方式组合初始化了两组共8幅不同编号的纹理。前4幅纹理都用同一幅32像素×32像素的图片生成，后4幅纹理都用同一幅256像素×256像素的图片生成，每组中4幅在MIN和MAG情况下采用了不同的采样方式组合。

（4）完成了onSurfaceCreated方法的开发后，下面开发的是加载纹理的initTexture方法，其代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_3/com/bn/Sample7_3目录下的MySurfaceView.java。

```
public int initTexture(int drawableId,float sample1,float sample2){
    int[] textures = new int[1];                         //创建一个数组
    GLES20.glGenTextures(
            1,                                           //产生的纹理 id 的数量
            textures,                                    //纹理 id 的数组
            0                                            //偏移量
    );
    int textureId=textures[0];                           //纹理数组
    GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureId);//绑定纹理
        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,     //确定纹理采样方式
                GLES20.GL_TEXTURE_MIN_FILTER,sample1);
        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,     //确定纹理采样方式
                GLES20.GL_TEXTURE_MAG_FILTER,sample2);
        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
                GLES20.GL_TEXTURE_WRAP_S,GLES20.GL_CLAMP_TO_EDGE);
        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
                GLES20.GL_TEXTURE_WRAP_T,GLES20.GL_CLAMP_TO_EDGE);
       //通过输入流加载图片
       InputStream is = this.getResources().openRawResource(drawableId);
       Bitmap bitmapTmp;                                 //声明 Bitmap 的引用
       try {
        bitmapTmp = BitmapFactory.decodeStream(is); //获取 Bitmap 的对象
       }
       finally {  try {is.close();}catch(IOException e) {e.printStackTrace();}}//关闭流
       }
       //实际加载纹理
       GLUtils.texImage2D(
```

```
28            GLES20.GL_TEXTURE_2D, //纹理类型
29            0,                    //纹理的层次，0表示基本图像层，可以理解为直接贴图
30            bitmapTmp,            //纹理图像
31            0                     //纹理边框尺寸
32        );
33        bitmapTmp.recycle();      //纹理加载成功后释放图片
34        return textureId;
35    }
```

- initTexture方法有3个参数：drawableId、sample1、sample2，其中sample1为MIN情况的采样方式，sample2为MAG的采样方式。采样方式由传入的参数决定，而不是固化在方法代码中，进一步提高了灵活性。
- 第9-17行首先绑定了纹理，然后根据sample1与sample2参数设置了纹理的采样方式，同时设置纹理的拉伸方式为截取。
- 第18-33行为通过输入流将纹理图加载进内存，然后将纹理图加载进显存并释放内存中的复制。

> **提示**
>
> 该案例中使用的顶点着色器和片元着色器与前面7.1节中纹理三角形案例使用的顶点着色器与片源着色器代码完全相同，这里不再赘述，需要的读者可以自行查看随书光盘中的源代码。

7.4 mipmap纹理技术

有一些经验的开发人员都知道，当需要处理的场景很大时（如一大片铺满相同纹理的丘陵地形），若不采用一些技术手段，可能会出现远处地形视觉上更清楚，近处地形更模糊的反真实现象。这主要是

由于透视投影中有近大远小的效果，远处的地形投影到屏幕上尺寸比较小，近处的尺寸比较大，而整个场景使用的是同一幅纹理图。

因此对远处的山体而言纹理图被缩小进行映射，自然很清楚；而近处的山体可能纹理图需要被拉大进行映射，自然就发虚。聪明的读者可能会想到，应该对远处的地形采用尺寸较小分辨率低的纹理，近处的采用尺寸较大分辨率高的纹理，这其实就是mipmap的基本思想。

提示

mipmap这个术语最早出现于1983年，由Lance williams在其论文“Pyramidal Parametrics”中首次提出。mip表示拉丁语“multum in parvo”，含义为“一块小地方有很多东西”，map是mapping的缩写，表示纹理映射。

不过在应用中若要自己开发根据场景视觉大小自动选择恰当分辨率的纹理进行映射的功能会非常复杂，幸运的是mipmap仅需要在加载纹理时进行一些设置即可，其他的工作是由渲染管线自动完成的。mipmap的基本工作原理如图7-23所示。

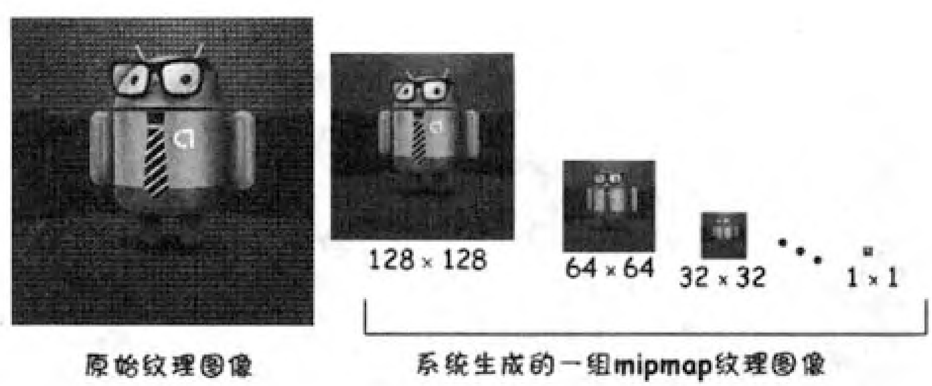


图7-23　mipmap纹理图像

从图7-23中可以看出，开发人员只需要提供一幅原始纹理图，系统会在纹理加载时自动生成一系列由大到小的纹理图。每幅纹理图是前一幅尺寸的1/2，直至纹理图的尺寸缩小到1×1。一系列纹理图中的第一副就是原始纹理图，因此可以轻松地计算出，一系列的mipmap纹理图占用的空 间接近原始纹理图的2倍。

一系列的mipmap纹理图生成成功后，当应用程序运行时，渲染管线会首先根据情况计算出细节级别，然后根据细节级别决定使用系列中哪一个分辨率的纹理图。开发中将采样方式设置为mipmap并自动生成一系列的mipmap纹理图的基本代码如下。

```
GLES20.glTexParameteri ( GLES20.GL_TEXTURE_2D, //设置MIN情况为mipmap最近点采样
GLES20.GL_TEXTURE_MIN_FILTER, GLES20.GL_LINEAR_MIPMAP_NEAREST);
GLES20.glTexParameteri ( GLES20.GL_TEXTURE_2D, //设置MAG情况为mipmap线性采样
GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_LINEAR_MIPMAP_LINEAR);
GLES20.glGenerateMipmap(GLES20.GL_TEXTURE_2D);//自动生成mipmap系列纹理
```

提示

关于mipmap技术的具体使用，将会在后继章节中介绍灰度图地形的部分详细介绍，并给出案例，这里读者仅需要了解基本概念、思路即可。

7.5　多重纹理与过程纹理

本节将通过一个案例向读者介绍从OpenGL ES 2.0才开始支持的两个高级特性，多重纹理与过程纹理。这两个特性在固定渲染管线的OpenGL ES 1.x中是无法实现的，有了这两个特性场景的真实感会大大提高。

7.5.1　案例概览

本章前面给出的所有案例中，对同一图元都只采用了一幅纹理图，这在有些情况下就显得不够强大了。本节将给出一个对同一图元采用多幅纹理图的案例，其运行效果如图7-24和图7-25所示。

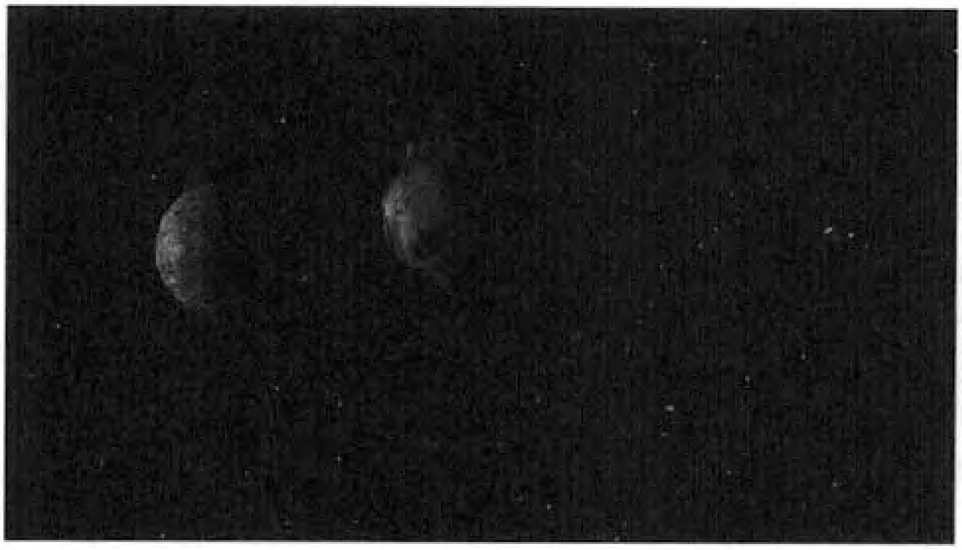


图7-24　运行效果图1

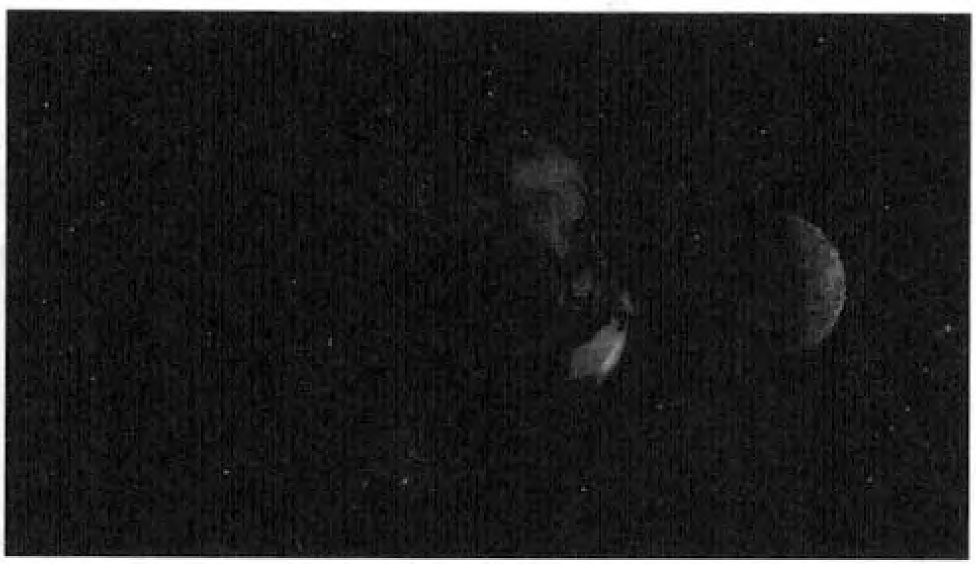


图7-25　运行效果图2

从图7-24和图7-25中可以看出，整个案例展示的是地月系在星空中的场景，中间的地球与前面的案例不同，其纹理不是固定的。阳光照耀到的区域使用的是白天的纹理（如图7-26所示），阳光没有照耀到的区域使用的是夜晚万家灯火的纹理（如图7-27所示），在白天和黑夜的边缘是平滑 过渡的。

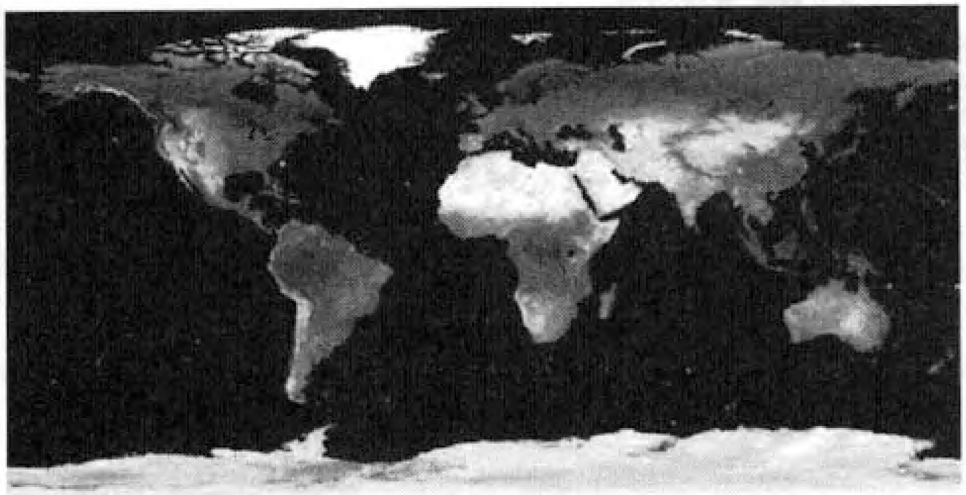


图7-26　白天的地球纹理



图7-27　黑夜的地球纹理

可以明显地感觉到，此案例的场景真实度比前面案例有了很大的提高。

- 对同一个图元采用多幅纹理图，这种技术称之为多重纹理。
- 在多重纹理变化的边界根据某种规则进行平滑过渡，这种技术称之为过程纹理。这种平滑过渡在很多情况下都会用到，如本案例中的白天纹理与黑夜纹理的过渡，丘陵地形中根据海拔不同进行的纹理过渡等。

提示

由于单色灰度印刷的原因，效果图可能看起来不是很清楚。这时可以参考本书前面的彩页或使用真机运行观察。同时，本案例中的阳光方向是可以随着手指在屏幕上的左右滑动而变化的。

7.5.2　将2D纹理映射到球面上的策略

本章前面给出的纹理映射相关案例都是将2D平面纹理映射到3D空间中的平面上，而从上一小节的运行效果图中可以看出本节的案例是将2D平面纹理映射到了三维的球面上。通过前面一章中光照球体的案例，读者应该已经了解到OpenGL ES 3D空间中的球面实际上是由一个一个独立的三角形组成的。

因此本案例中纹理映射的关键点就是如何产生这些小三角形中每个顶点对应的纹理坐标，这就比从平面纹理到平面的映射要复杂一些了。这种映射需要采用一些技术手段，基本思路如图7-28所示。

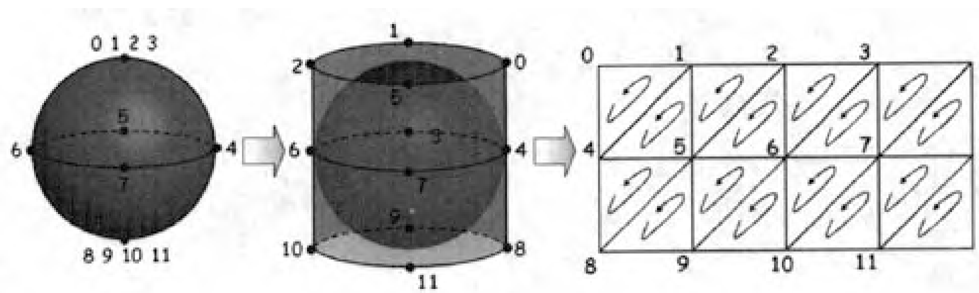


图7-28　三维球面展开过程示意图

提示

图7-28中为了画示意图方便，仅将球面在纬度方向切割成了两份，经度方向切割成了4份，实际开发中应该按照同样的思路多分割几份，这样球面就较为平滑了。

从图7-28中可以看出，首先对球面进行拓扑变换，将上下两极破开，再像橡皮膜一样拉伸，就可以将球面变换为圆柱面。然后对圆柱面进一步进行拓扑变换，将圆柱沿某一条竖直棱切开，并展开成一个

矩形。由于球面原来的顶点组成了一个一个的小三角形，因此展开的矩形也由一个一个小三角形组成。

说明

拓扑变换是拓扑几何中的一种变换，拓扑变换前后的两个图形是拓扑全等的。简单来说拓扑变换就是在不产生新顶点以及不改变顶点与顶点之间边的连接情况的前提下，任意地将顶点移动，这时连接这些顶点的边也可能被相应地拉伸、缩短、移位、旋转。在开发3D应用程序时，恰当运用拓扑变换辅助思考可以将问题大大地简化。

球面被展开成矩形后就很容易与2D的矩形纹理图进行匹配了，具体情况如图7-29所示。

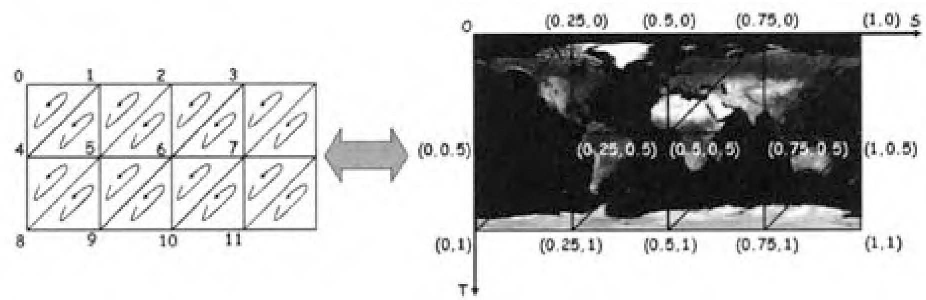


图7-29　三维球面的顶点与2D纹理的对应关系图

从图7-29中可以看出，根据矩形中每个顶点对应的S轴、T轴的位置可以非常方便地计算出每个顶点的纹理坐标。而矩形里面的顶点都是通过拓扑变换来自于球面上的顶点，与球面上的顶点一一对应，因此球面上每个顶点的纹理坐标就很自然地计算出来了。

球面上每个顶点都有了其对应的纹理坐标后，应用程序一运行就可以在屏幕上渲染出如前面图7-24与图7-25所示具有逼真视觉效果的地月系场景了。

提示

不但将2D纹理映射到球面上可以采用本小节介绍的策略，将2D纹理映射到其他形状立体物体的表面时也可以采用这种思维方式。

7.5.3 案例的场景结构

从前面的图7-24与图7-25中可以看出，本案例展现的是一个地月系的场景。其中地球位于场景中央，月球绕地球公转，地球和月球都有自转。场景中的变换组合使用了基本变换中的平移与旋转变换，细节如下所列。

- 首先，初始情况下地球与月球都是贴好纹理的、球心位于坐标原点的球，如图7-30所示。

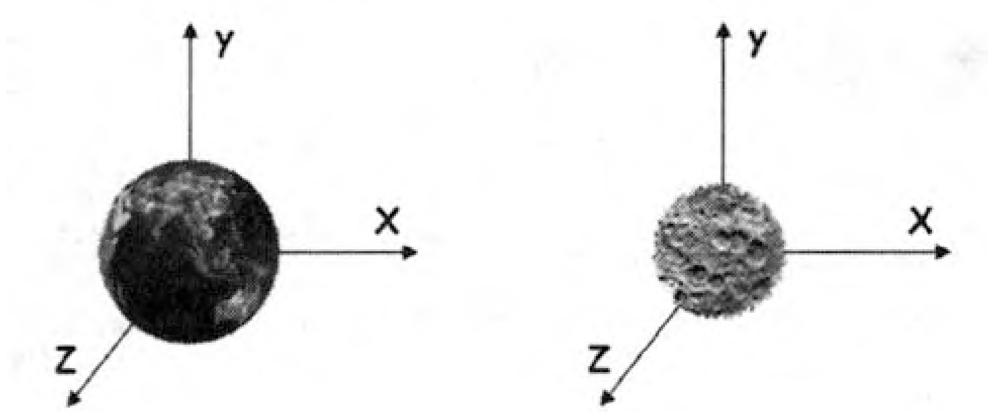


图7-30 原始情况下的地球及月球

- 接着在绘制每一帧画面时首先将坐标系绕Y轴旋转一定的角度（这个角度是一个变量，绘制每一帧时都会有微小的变化，代表地球自转）后绘制地球。
- 然后将坐标系沿x轴正方向推移一定的距离L（模拟地月间距）后，再将坐标系绕Y轴旋转一 定的角度（这个角度是一个变量，绘制每一帧时都会有微小的变化，代表月球的自转）后绘制月球。

通过如上的顺序变换坐标系、绘制地球、月球就会产生所期望的公转、自转效果，具体情况如图7-31所示。

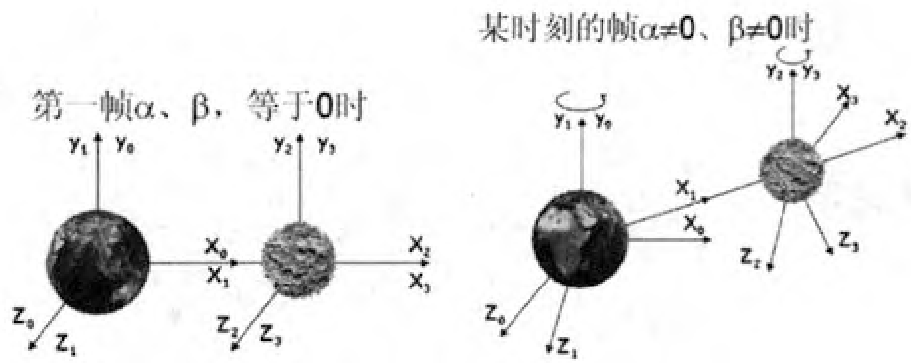


图7-31　每帧画面涉及的变换

> **说明**
>
> 0号坐标系为原始坐标系。1号坐标系为0号坐标系绕Y轴旋转α度后的坐标系，此时绘制地球。2号坐标系为将1号坐标系沿X轴推移距离L后的坐标系。3号坐标系为将2号坐标系绕Y轴旋转β度时的坐标系，此时绘制月球。最终，给观察者的感觉就是地球自转，月球绕地球公转的同时自己自转。

7.5.4　开发过程

前面已经介绍了本案例的运行效果与基本思路，接下来将介绍本案例的具体开发过程，主要包括着色器的开发与Java代码的开发两个阶段。

1．着色器的开发

本案例中的着色器一共有3套：星空着色器、月球着色器以及地球着色器，具体的开发步骤如下。

（1）首先开发的是用于绘制星空的顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_4/assets目录下的vertex_xk.sh文件。

```
uniform mat4 uMVPMatrix;                          //总变换矩阵
uniform float uPointSize;                         //点尺寸
attribute vec3 aPosition;                         //顶点位置
void main(){
  gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点的位置
  gl_PointSize=uPointSize;                        //点尺寸
}
```

- 此着色器功能并不复杂，主要是首先根据接收到的顶点位置及变换矩阵计算出顶点的最终绘制位置并传递给管线，接着将接收到的点绘制尺寸传递给管线。所谓传递给管线就是给恰当的内建变量赋值，如上述代码中的gl_Position、gl_PointSize等。
- 由于星星点的尺寸是从外界接收的，因此可以根据参数绘制出大小不同的星星，大大增加了场景的真实感。

（2）完成了用于绘制星空的顶点着色器的开发后，下面将开发用于绘制星空的片元着色器，其 代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_4/assets目录下的frag_xk.sh文件。

```
precision mediump float;                          //给出浮点默认精度
void main(){                                      // main 方法
  gl_FragColor = vec4(1.0,1.0,1.0,1.0);          //给此片元赋颜色值
}
```

说明

星空片元着色器的代码非常简单，仅仅是将固定的颜色传递给gl_FragColor内建变量，供管线绘制星空时使用。

（3）开发完了用于绘制星空的着色器后，下面将开发用于绘制月球的着色器。首先开发的是用于绘制月球的顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_4/assets目录下的vertex_moon.sh文件。

```
uniform mat4 uMVPMatrix;                               //总变换矩阵
uniform mat4 uMMatrix;                                 //变换矩阵
uniform vec3 uCamera;                                  //摄像机位置
uniform vec3 uLightLocationSun;                        //太阳光源位置
attribute vec3 aPosition;                              //顶点位置
attribute vec2 aTexCoor;                               //顶点纹理坐标
attribute vec3 aNormal;                                //法向量
varying vec2 vTextureCoord;                            //用于传递给片元着色器的纹理坐标信息
varying vec4 vAmbient;                                 //用于传递给片元着色器的环境光信息
varying vec4 vDiffuse;                                 //用于传递给片元着色器的散射光信息
varying vec4 vSpecular;                                //用于传递给片元着色器的镜面反射光信息
void pointLight(in vec3 normal, inout vec4 ambient, inout vec4 diffuse,
  inout vec4 specular, in vec3 uLightLocation,in vec4 lightAmbient,
  in vec4 lightDiffuse,in vec4 lightSpecular)          //定位光光照计算的方法
  {.....//该方法在第 6 章已详细介绍，这里不再赘述，读者可以自行查看随书光盘中的源代码}
void main(){
   gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点位置
   vec4 ambientTemp=vec4(0.0,0.0,0.0,0.0);          //初始化环境光 ambientTemp
   vec4 diffuseTemp=vec4(0.0,0.0,0.0,0.0);          //初始化散射光 diffuseTemp
   vec4 specularTemp=vec4(0.0,0.0,0.0,0.0);         //初始化镜面反射光 specularTemp
   pointLight(normalize(aNormal),ambientTemp,diffuseTemp,specularTemp,
    uLightLocationSun,vec4(0.05,0.05,0.025,1.0),vec4(1.0,1.0,0.5,1.0),vec4
    (0.3,0.3,0.15,1.0));
   vAmbient=ambientTemp;                            //将环境光的计算结果赋给 vAmbient
   vDiffuse=diffuseTemp;                            //将散射光的计算结果赋给 vDiffuse
   vSpecular=specularTemp;                          //将镜面反色光的计算结果赋给 vSpecular
   vTextureCoord=aTexCoor;                          //将顶点的纹理坐标传给片元着色器
}
```

说明

此着色器与上一章介绍光照时综合案例的顶点着色器代码非常类似，功能为给每个顶点计算出环境光、散射光、镜面反射光分量并通过易变变量传递给片元着色器。最大的区别就是，增加了将纹理坐标传递给片元着色器的功能。

（4）开发完了用于绘制月球的顶点着色器后，下面开发的是用于绘制月球的片元着色器，其具体代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_4/assets目录下的frag_moon.sh文件。

```
precision mediump float;                        //给出浮点默认精度
varying vec2 vTextureCoord;                     //接收从顶点着色器过来的顶点纹理坐标
varying vec4 vAmbient;                          //接收从顶点着色器过来的环境光信息
varying vec4 vDiffuse;                          //接收从顶点着色器过来的散射光信息
varying vec4 vSpecular;                         //接收从顶点着色器过来的镜面反射光信息
uniform sampler2D sTexture;                     //纹理内容数据
void main() {
  vec4 finalColor = texture2D(sTexture, vTextureCoord);  //给此片元从纹理中采样出颜色值
  gl_FragColor = finalColor*vAmbient+finalColor*vSpecular+finalColor*vDiffuse;
                                                //给此片元颜色值
}
```

说明

该片元着色器与本章前面案例中的片元着色器最大的区别就是，不仅根据易变变量传入的纹理坐标进行了纹理采样，还将纹理

采样的结果与传入的环境光、散射光、镜面反射光分量进行加权计算，最终得出片元在光照与纹理协同作用下的颜色值。

（5）完成了月球着色器的开发后，下面将开发的是用于绘制地球的着色器。首先开发的是用于绘制地球的顶点着色器，其代码与前面步骤3中用于绘制月球的顶点着色器代码基本相同，这里不再赘述，需要的读者可以自行参考光盘中的源代码。

（6）开发完了用于绘制地球的顶点着色器后，下面接着要开发的是用于绘制地球的具有多重纹理、过程纹理功能的片元着色器，其代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_4/assets目录下的frag_earth.sh文件。

```
1 precision mediump float;                 //给出浮点默认精度
2 varying vec2 vTextureCoord;              //接收从顶点着色器过来顶点纹理坐标信息
3 varying vec4 vAmbient;                   //接收从顶点着色器过来环境光最终强度
4 varying vec4 vDiffuse;                   //接收从顶点着色器过来散射光最终强度
5 varying vec4 vSpecular;                  //接收从顶点着色器过来镜面光最终强度
6 uniform sampler2D sTextureDay;           //白天纹理的内容数据
7 uniform sampler2D sTextureNight;         //夜晚纹理的内容数据
8 void main(){
9   vec4 finalColorDay;                    //从白天纹理中采样出颜色值
10      vec4 finalColorNight;              //从夜晚纹理中采样出颜色值
11      finalColorDay= texture2D(sTextureDay, vTextureCoord);  //采样出白天纹理的颜色值
12      finalColorDay = finalColorDay*vAmbient+finalColorDay*vSpecular+finalColorDay*
        vDiffuse;
13      finalColorNight = texture2D(sTextureNight, vTextureCoord);//采样出夜晚纹理的颜色值
14      finalColorNight = finalColorNight*vec4(0.5,0.5,0.5,1.0);//计算出的该片元夜晚颜色值
15      if(vDiffuse.x>0.21){               //当散射光分量大于 0.21 时
16        gl_FragColor=finalColorDay;      //采用白天纹理
17      }else if(vDiffuse.x<0.05){         //当散射光分量小于 0.05 时
18        gl_FragColor=finalColorNight;         //采用夜间纹理
19      }else{        //当环境光分量大于 0.05 小于 0.21 时，为白天夜间纹理的过渡阶段
20        float t=(vDiffuse.x-0.05)/0.16;       //计算白天纹理应占纹理过渡阶段的百分比
21        gl_FragColor=t*finalColorDay+(1.0-t)*finalColorNight;//计算白天黑夜过渡阶段的颜色值
22   }}
```

- 第2-5行声明了用来接收顶点着色器传过来的纹理坐标、环境光、散射光、镜面反射光分量的易变变量。

- 第6-7行声明了两个sampler2D类型的变量，分别用来接收白天与黑夜的纹理，这是此片元着色器与本章其他案例中片元着色器的一个很大的不同点。
- 第11-12行首先根据接收的纹理坐标对白天的纹理进行了采样，然后根据接收的环境光、散射光、镜面反射光分量对采样值进行了加权计算，最后得到了片元若为白天时的颜色值。
- 第13-14行首先根据接收的纹理坐标对黑夜的纹理进行了采样，然后对黑夜的纹理亮度进行变化后得到了片元若为黑夜时的颜色值。
- 第15-22行根据此片元散射光的亮度将片元的最终颜色分3种情况计算：若散射光亮度大于0.21，则此片元完全采用白天时的颜色值；若散射光亮度小于0.05，则此片元完全采用黑夜时的颜色值；若散射光亮度介于0.05～0.21，则根据亮度将白天与黑夜的颜色值进行加权计算产生最终的片元颜色值。这就是所谓的过程纹理技术，采用其处理后过渡是平滑的，视觉效果更自然、真实。

2．Java代码的开发

完成了所有着色器的开发后，就可以进行Java部分代码的开发了，具体步骤如下。

（1）首先开发的是用于绘制星空天球的Celestial类，其作用为在茫茫宇宙中显现群星闪烁的视觉效果，具体代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_4/com/bn/Sample7_4目录下的Celestial.java。

```
package com.bn.Sample7_4;                                    //包声明
import static com.bn.Sample7_4.ShaderUtil.createProgram;    //相关类的引入
……//此处省略部分类引入的代码，请读者自行查看随书光盘的源代码
import android.opengl.GLES20;                                //相关类的引入
public class Celestial {                                     //表示星空天球的类
    final float UNIT_SIZE=10.0f;                             //天球半径
    private FloatBuffer   mVertexBuffer;                     //顶点坐标数据缓冲
    int vCount=0;                                            //星星数量
    float yAngle;                                            //天球绕 Y 轴旋转的角度
        float scale;                                         //星星尺寸
        String mVertexShader;                                //顶点着色器代码脚本
        String mFragmentShader;                              //片元着色器代码脚本
        int mProgram;                                       //自定义渲染管线着色器程序 id
        int muMVPMatrixHandle;                               //总变换矩阵引用
        int maPositionHandle;                                //顶点位置属性引用
        int uPointSizeHandle;                                //顶点尺寸参数引用
        public Celestial(float scale,float yAngle,int vCount,MySurfaceView mv){
         this.yAngle=yAngle;
```

```
        this.scale=scale;                   //绕 Y 轴旋转的角度
        this.vCount=vCount;                 //顶点的数量
        initVertexData();                   //调用初始化顶点数据的 initVertexData 方法
        initShader(mv);                     //调用初始化着色器的 initShader 方法
    }
    public void initVertexData(){          //自定义初始化顶点数据的 initVertexData 方法
     //顶点坐标数据的初始化
        float vertices[]=new float[vCount*3];
        for(int i=0;i<vCount;i++){
          double angleTempJD=Math.PI*2*Math.random();      //随机产生一个方位角
          double angleTempWD=Math.PI*(Math.random()-0.5f);//随机产生一个仰角
          //根据仰角及方位角计算每个星星的 x、y、z 坐标
          vertices[i*3]=(float)(UNIT_SIZE*Math.cos(angleTempWD)*Math.sin(angleTempJD));
          vertices[i*3+1]=(float)(UNIT_SIZE*Math.sin(angleTempWD));
          vertices[i*3+2]=(float)(UNIT_SIZE*Math.cos(angleTempWD)*Math.cos(angleTempJD));
        }
        //创建顶点坐标数据缓冲
        ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
        vbb.order(ByteOrder.nativeOrder());        //设置字节顺序为本地操作系统顺序
        mVertexBuffer = vbb.asFloatBuffer();       //转换为 int 型缓冲
        mVertexBuffer.put(vertices);               //向缓冲区中放入写入数据
        mVertexBuffer.position(0);                 //设置缓冲区起始位置
    }
    public void initShader(MySurfaceView mv){/*代码省略，需要的读者请自行参考随书光盘*/}
    public void drawSelf(){
         GLES20.glUseProgram(mProgram);            //制定使用某套着色器程序
        //将最终变换矩阵传入着色器程序
        GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, false, MatrixState.
        getFinalMatrix(), 0);
        GLES20.glUniform1f(uPointSizeHandle, scale);  //将顶点尺寸传入着色器程序
        GLES20.glVertexAttribPointer(              //将顶点位置数据传送进渲染管线
              maPositionHandle,
              3,
              GLES20.GL_FLOAT,
              false,
              3*4,
              mVertexBuffer
        );
        //启用顶点位置数据
        GLES20.glEnableVertexAttribArray(maPositionHandle);
        GLES20.glDrawArrays(GLES20.GL_POINTS, 0, vCount);      //绘制星星点
}}
```

- 第17-23行为Celestial类的构造器，主要工作为接收并保存一些成员变量的值，调用initVertexData方法初始化顶点数据，调用initShader方法初始化着色器。
- 第24-41行为初始化顶点坐标数据的initVertexData方法，其在指定半径的天球上随机生成了指定个数的星星。每产生一颗星星时，先随机给出星星的仰角及方位角，然后根据解析几何公式计算出此星星的x、y、z 坐标。
- 第42行为初始化着色器的initShader方法，该方法的代码与前面案例的基本一致，都是首先创建着色器程序，然后从着色器程序中获得需要的引用值并保存，需要的读者请参考随书光盘。
- 第43-59行为绘制星空天球的drawSelf方法，该方法中大部分与前面的案例一致，最大的区别就是第58行的绘制方式不再是GLES20.GL_TRIANGLES，而是GLES20.GL_POINTS。这是由于星空是由一个一个独立的点构成的，而不是由三角形构成的。

（2）完成了Celestial类的开发后，下面将要开发的是采用多重纹理的Earth类。首先开发的是该类的代码框架，具体代码如下。



代码位置：见随书光盘中源代码/第7章/Sample7_4/com/bn/Sample7_4目录下的Earth.java。

```
1 package com.bn.Sample7_4;                                       //包声明
2 import static com.bn.Sample7_4.ShaderUtil.createProgram; //相关类的引入
3 ……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
4 import android.opengl.GLES20;                                   //相关类的引入
5 //表示地球的类，采用多重纹理
6 public class Earth {
7     int mProgram;                                               //自定义渲染管线着色器程序id
8     ……//省略了一些常用的成员变量声明，与前面的案例套路基本相同，需要的读者参看光盘
9     FloatBuffer   mTexCoorBuffer;                               //顶点纹理坐标数据缓冲
10        int uDayTexHandle;                                      //白天纹理属性引用
11        int uNightTexHandle;                                    //黑夜纹理属性引用
12        int vCount=0;                                           //顶点数量
13        public Earth(MySurfaceView mv,float r){
14         initVertexData(r);                        //调用初始化顶点数据的initVertexData方法
15         initShader(mv);                           //调用初始化着色器的intShader方法
16        }
17        public void initVertexData(float r){/*代码省略，需要的读者请参考随书光盘*/}
18        public void initShader(MySurfaceView mv){ /*代码省略，需要的读者请参考随书光盘*/}
19        public void drawSelf(int texId,int texIdNight){ /*代码省略，后面详细介绍*/}
20        public float[] generateTexCoor(int bw,int bh){ /*代码省略，后面详细介绍*/}
21    }
```

- 第7-12行定义了一些程序中需要用到的成员变量，具有特殊性的就是此3D物体有两个纹理引用：uDayTexHandle与uNightTexHandle。这也对应到前面地球片元着色器中有两个sampler2D类型的纹理变量，分别用来接收白天与黑夜的纹理。
- 第26-27行为初始化顶点数据的initVertexData方法与初始化着色器的initShader方法，这两个方法与本书前面章节的案例基本一致，主要是将绘制需要的各项顶点信息放进相应的缓冲，如顶点位置、法向量、纹理坐标等。因此不再赘述。

• 第28-29行的两个方法在后面的步骤中详细介绍，这里只给出了方法签名。

（3）完成了Earth类代码框架的开发后，下面将开发根据参数自动计算顶点纹理坐标的generateTexCoor方法，其代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_4/com/bn/Sample7_4目录下的Earth.java。

```
public float[] generateTexCoor(int bw,int bh){  //自动切分纹理产生纹理数组的方法
```

```
    float[] result=new float[bw*bh*6*2];
    float sizew=1.0f/bw;                        //列数
    float sizeh=1.0f/bh;                        //行数
    int c=0;
    for(int i=0;i<bh;i++){                      //循环
        for(int j=0;j<bw;j++){                  //循环
            //每行列一个矩形，由两个三角形构成，共6个点，12个纹理坐标
            float s=j*sizew;
                float t=i*sizeh;                //得到i行j列小矩形的左上点的纹理坐标值
                result[c++]=s;
                result[c++]=t;                  //该矩形左上点纹理坐标值
                result[c++]=s;
                result[c++]=t+sizeh;            //该矩形左下点纹理坐标值
                result[c++]=s+sizew;
                result[c++]=t;                  //该矩形右上点纹理坐标值
                result[c++]=s+sizew;
                result[c++]=t;                  //该矩形右上点纹理坐标值
                result[c++]=s;
                result[c++]=t+sizeh;            //该矩形左下点纹理坐标值
                result[c++]=s+sizew;
                result[c++]=t+sizeh;            //该矩形右下点纹理坐标值
        }}
        return result;                          //返回结果
}
```

- 从上述代码中可以看出，此方法根据接收的参数bw、bh，将一幅纹理图看作切割成bh行、bw列的bwxbh个小方块。然后在循环中依次将每个小方块4个顶点的纹理坐标按照一定的卷绕方式存放到数组中返回，这实际就是前面7.5.2策略的实现。

- 读者可能会发现每个小方块对应了6对纹理坐标，这是因为本案例采用了GL_TRIANGLES绘制方式。将每个矩形看作由两个三角形组成，每个三角形有3个顶点，因此一共有6个顶点。这显然会在每个小方块中有两个重复的顶点，如果读者不希望有这些重复的顶点，可以修改程序的相关部分，改成使用GL_TRIANGLE_STRIP绘制方式。

（4）完成了根据切割的行列数生成纹理坐标序列的generateTexCoor方法的开发后，下面开发的是用于绘制地球的drawSelf方法，其具体代码如下。

代码位置：见随书光盘中源代码/第7章/Sample7_4/com/bn/Sample7_4目录下的Earth.java。

```
public void drawSelf(int texId,int texIdNight) {
    GLES20.glUseProgram(mProgram);                 //指定使用某套着色器程序
   //将最终变换矩阵传入着色器程序
  GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, false, MatrixState.getFinalMatrix(), 0);
   //将位置、旋转变换矩阵传入着色器程序
   GLES20.glUniformMatrix4fv(muMMatrixHandle, 1, false, MatrixState.getMMatrix(), 0);
   //将摄像机位置传入着色器程序
   GLES20.glUniform3fv(maCameraHandle, 1, MatrixState.cameraFB);
   //将光源位置传入着色器程序
      GLES20.glUniform3fv(maSunLightLocationHandle, 1, MatrixState.lightPositionFBSun);
       …… //省略了传送顶点位置、顶点纹理坐标、顶点法向量的代码，需要的读者请参考随书光盘
```

```
12      //允许相关的数据数组
13      GLES20.glEnableVertexAttribArray(maPositionHandle); //顶点位置
14      GLES20.glEnableVertexAttribArray(maTexCoorHandle);  //顶点纹理
15      GLES20.glEnableVertexAttribArray(maNormalHandle);   //顶点法向量
16      //绑定纹理
17      GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
18      GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, texId);  //白天纹理
19      GLES20.glActiveTexture(GLES20.GL_TEXTURE1);
20      GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, texIdNight); //黑夜纹理
21      GLES20.glUniform1i(uDayTexHandle, 0);               //通过引用指定白天纹理
22      GLES20.glUniform1i(uNightTexHandle, 1);             //通过引用指定黑夜纹理
23      GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount); //以三角形方式执行绘制
24  }
```

- 此方法中大部分代码都与本书前面案例中的基本一致，因此做了一些省略，需要的读者请参考随书光盘。
- 此方法与前面案例最大的不同就是第17-22行的纹理相关操作：一方面其激活并绑定了两个纹理；一方面通过glUniformli方法给片元着色器中的两个sampler2D类型的变量指定了各自的纹理。第二个操作是以前没有的，这是因为当只有一个纹理时不会产生混淆，但有多于一个纹理的情况就需要采用类似的操作，避免系统无法判断哪个纹理与哪个sampler2D类型的变量匹配。

（5）最后需要开发的是用来渲染整个场景的MySurfaceView类，此类中的大部分代码与本书前面很多案例中的相同，因此这里只给出基于基本变换实现场景中地球、月球公转及自转效果的部分代码。

```
1 //绘制地球、月球
2 MatrixState.pushMatrix();                                   //保护现场
3 MatrixState.rotate(eAngle, 0, 1, 0);                        //地球自转
4 earth.drawSelf(textureIdEarth,textureIdEarthNight);         //绘制地球
5 MatrixState.transtate(2f, 0, 0);                            //推坐标系到月球位置
6 MatrixState.rotate(eAngle, 0, 1, 0);                        //月球自转
7 moon.drawSelf(textureIdMoon);                               //绘制月球
8 MatrixState.popMatrix();                                    //恢复现场
9 //绘制星空
10    MatrixState.pushMatrix();                               //保护现场
11    MatrixState.rotate(cAngle, 0, 1, 0);                    //星空天球旋转
12    cSmall.drawSelf();                                      //绘制小尺寸星星的天球
13    cBig.drawSelf();                                        //绘制大尺寸星星的天球
14    MatrixState.popMatrix();                                //恢复现场
```

- 第2-8行依照前面7.5.3小节介绍的变换、绘制顺序实现了给定具体参数值（地球、月球当前的自转角度）情况下一帧地月系场景的绘制。
- 第10-14行单独绘制了星空天球。从代码中可以看出星空天球有两个，分别用来显示不同尺寸的星星。同时星空天球的旋转是独立于地月系的，这也进一步增加了场景的真实感。

提示

本案例中还有其他一些类，如MyActivity、ShaderUtil等，与前面很多案例中的基本一致。另外还有一个名称为Moon的用来绘制月球的类，其与绘制地球的Earth类基本相同，只是没有采用多重纹理。因本书篇幅有限，这些类都不再赘述，需要的读者请参考随书光盘中的源代码。

7.6　本章小结

本章中主要介绍了OpenGL ES 2.0中纹理映射各方面的相关知识，包括纹理映射的基本原理、纹理拉伸、纹理采样、mipmap纹理、多重纹理与过程纹理等。通过本章的学习，读者应该能够从容应对实际开发中纹理相关的需求，为后面高级知识的学习打下良好的基础。

第8章　3D基本形状的构建

前面几章介绍了很多OpenGL ES 2.0的基础知识，但绘制的立体形状种类很少，除了立方体就是圆球，这是远远不够的。本章将进一步介绍几种常见立体形状的构建，包括圆柱、圆锥、圆环、螺旋管、几何球、足球碳、贝塞尔曲线及旋转面等。

提示

由于本章很多案例中的基本代码与前面章节很多案例中的基本一致，因此，在对本章案例进行介绍时，一些重复部分不再赘述，着重介绍本章有特色、有代表性的部分，需要的读者请自行查阅随书光盘中的源代码。

8.1　圆柱体

本节将介绍通过两种组件构建的圆柱体，这两种组件分别为圆面和柱面。首先介绍圆面和柱面顶点原始位置生成的基本原理，然后给出实现的案例Sample8_1。

8.1.1　顶点原始位置的生成

从前面的章节中已经了解到，OpenGL ES 2.0中任何立体物体实质上都是由多个小三角形组成的。自然构成圆柱的圆面与柱面也不例外，因此本小节将介绍如何将圆面与柱面切分成三角形，以及如何计算三角形每个顶点的坐标。

1．顶点坐标的计算

初始状态下，将圆面几何中心对到XOY平面的原点，则根据切分单位角度的不同可以方便地使用三角函数计算出三角形边缘顶点的*X* 与*Y* 坐标，如图8-1所示。

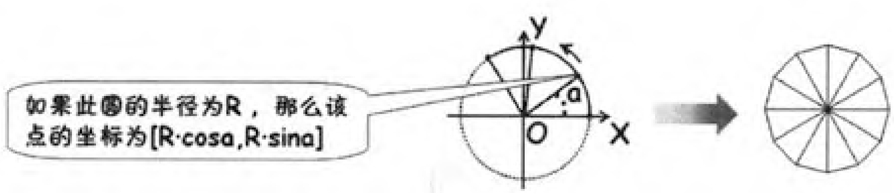


图8-1　圆面的形成和圆面顶点坐标的计算

根据图8-1计算出来的三角形边缘顶点的坐标不但可以用于构建圆面，其实也是构成柱面的顶 点。圆柱的侧面可以看做由一个矩形面（此矩形面沿着横边切分为n份）以空间中的某一条直线为轴（本案例中以*Y* 轴为轴）卷绕而成的，如图8-2所示。

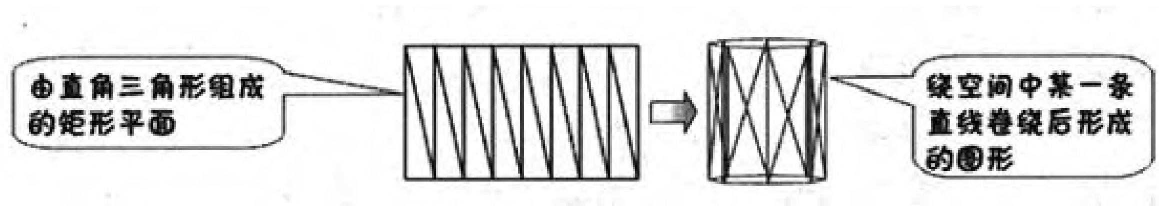


图8-2　圆柱侧面的示意图

提示

圆面上所有的顶点Z 坐标都相同，根据圆柱的长度、位置情况直接给出即可，这点可以参考后面的顶点坐标计算相关代码。

提示

图8-2中为了介绍的方便，将一个圆柱体侧面切分为8等份，随着切分份数的增加，所形成的图形将越来越接近圆柱面。

从图8-2中可以看出，若将圆柱的侧面切分为n份，且圆柱的半径为R，当前棱对应的点角度为a。那么由三角函数及空间解析几何可知，构成圆柱侧面的此直角三角形的顶点坐标如图8-3所示。

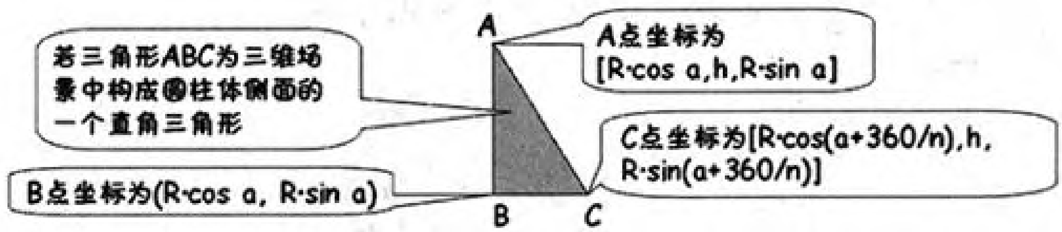


图8-3　圆柱侧面顶点坐标计算的示意图

说明

将构成圆柱体侧面的所有三角形顶点坐标按图8-3所示计算完成后，再按一定次序存放到数组中，便可以作为圆柱侧面的顶点坐标数据。

2．顶点纹理坐标与法向量的计算

前面介绍完了顶点坐标数据的计算，下面将进一步介绍顶点纹理坐标与法向量的计算。首先介绍圆面顶点纹理坐标的计算，如图8-4所示。

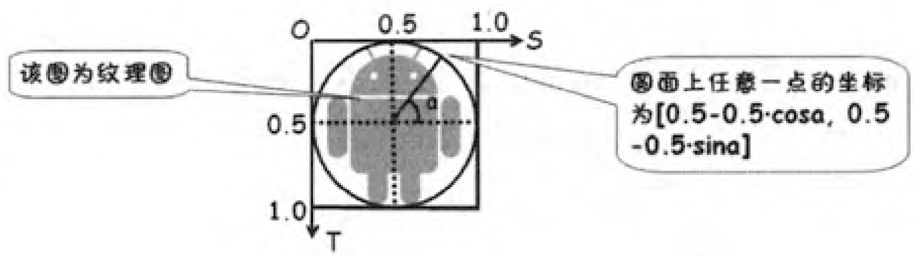


图8-4　圆面纹理坐标的计算

说明

将圆面的中心点对应到纹理坐标[0.5，0.5]的位置，那么圆面上任何一点的纹理坐标数据如图8-4所示，可通过三角函数方便地求得。

接着介绍柱面顶点纹理坐标的计算，设圆柱侧面切分的份数为n，将圆柱侧面展开形成一个矩 形平面，将纹理图左上角对应到展开后矩形的左上角，即可方便地计算出各顶点的纹理坐标，如图8-5所示。

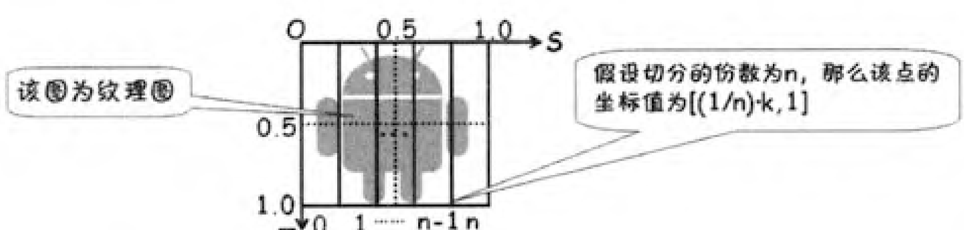


图8-5　圆柱侧面纹理坐标的计算

说明

图8-5中k为当前份数编号，取值范围为0~n。另外要注意的是，这里给出的纹理坐标计算方式是将纹理图拉伸以充满整个展开后的矩形平面的。

完成了顶点和纹理坐标计算方式的介绍后，下面介绍法向量的计算法，具体如下所列。

- 圆面上顶点的法向量其实都是相同的，垂直于圆面所在的平面。因此在圆面垂直于某坐标轴的情况下，直接给出即可，不需要计算。如本案例中垂直于Z轴，因此法向量都为[0，0，1]。
- 柱面上顶点的法向量与底面平行，垂直于柱面。因此法向量的一个分量为0，其他两个分量与顶点坐标值相同。如在本案例中柱面的轴为*Y* 轴，则法向量Y分量为0，X、Z分量与顶点坐标一致。

提示

读者可能会产生这样的困扰，圆面原始情况下是垂直于Z轴的，柱面是围绕着*Y* 轴的，这样也不能组装出圆柱体啊。是的，直接组装是不行的，但在组装时对圆面与柱面进行一下恰当的平移、旋转变换就可以了，详情见后面的代码部分。

8.1.2 案例的开发

了解了圆柱体两个部件的顶点坐标、纹理坐标、法向量的计算方法后，下面应该了解一下本节案例的运行效果了，如图8-6所示。

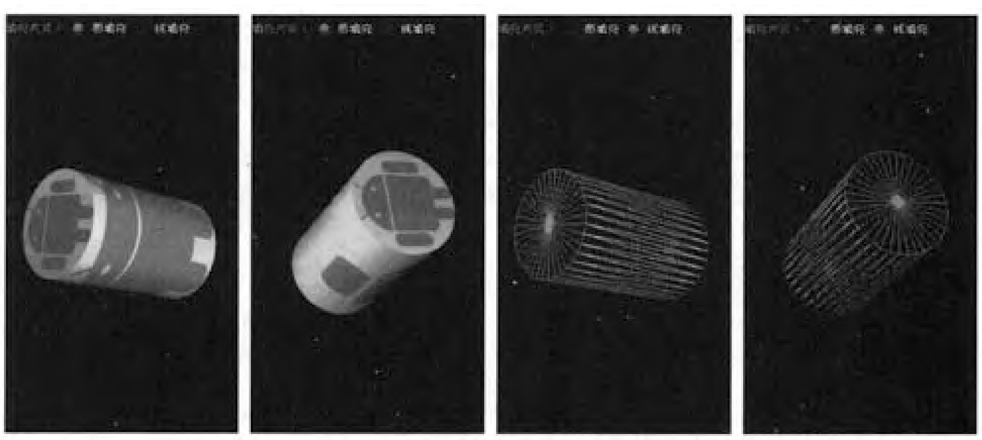


图8-6 Sample8_1运行效果图

说明

从图8-6中可以看出，本案例提供了两种绘制方式：面填充和线段。左侧两幅图为面填充方式下本案例的运行效果图，右侧两幅图为线段方式下本案例的运行效果图。

了解了本案例的运行效果后，就可以进行代码的开发了。这里仅给出本案例中有代表性的圆面、柱面的顶点坐标、纹理坐标、法向量生成以及将圆面、柱面组装成圆柱体的相关代码，具体如下所列。

（1）首先给出的是圆面顶点坐标、纹理坐标和法向量生成的相关代码，具体内容如下。

代码位置：见随书光盘中源代码/第8章/Sample8_1/com/bn/Sample8_1目录下的Circle.java。

```
float[] vertices=new float[vCount*3];                          //顶点坐标数组
float[] textures=new float[vCount*2];                          //顶点纹理坐标数组
int count=0;                                                   //顶点坐标的计数器
int stCount=0;                                                 //顶点纹理坐标的计数器
for(float angdeg=0;Math.ceil(angdeg)<360;angdeg+=angdegSpan){//生成每个三角形的顶点数据
 double angrad=Math.toRadians(angdeg);                         //当前弧度
 double angradNext=Math.toRadians(angdeg+angdegSpan);          //下一弧度
 vertices[count++]=0; vertices[count++]=0; vertices[count++]=0; //圆面中心点的顶点坐标
 textures[stCount++]=0.5f;  textures[stCount++]=0.5f;     //圆面中心点的顶点纹理坐标
     vertices[count++]=(float) (-r*Math.sin(angrad));    //当前弧度对应的边缘顶点坐标
     vertices[count++]=(float) (r*Math.cos(angrad));
     vertices[count++]=0;
     textures[stCount++]=(float) (0.5f-0.5f*Math.sin(angrad));
                                                          //当前弧度对应的边缘顶点纹理坐标
     textures[stCount++]=(float) (0.5f-0.5f*Math.cos(angrad));
     vertices[count++]=(float) (-r*Math.sin(angradNext));//下一弧度对应的边缘顶点坐标
     vertices[count++]=(float) (r*Math.cos(angradNext));
     vertices[count++]=0;
     textures[stCount++]=(float) (0.5f-0.5f*Math.sin(angradNext));
                                                          //下一弧度对应的边缘顶点纹理坐标
     textures[stCount++]=(float) (0.5f-0.5f*Math.cos(angradNext));
    }
    float[] normals=new float[vertices.length];          //顶点法向量数据数组
    for(int i=0;i<normals.length;i+=3){
        normals[i]=0;  normals[i+1]=0;  normals[i+2]=1;
    }
```

说明

上述代码根据上一小节介绍的原理以一定的角度间距（angdegSpan）循环计算出了组成圆面的小三角形顶点的各项数据，包括顶点坐标、纹理坐标、法向量。

（2）接着给出柱面顶点坐标、纹理坐标和法向量生成的相关代码，具体内容如下。

代码位置：见随书光盘中源代码/第8章/Sample8_1/com/bn/Sample8_1目录下的CylinderSide.java。

```
float[] vertices=new float[vCount*3];                    //顶点法向量数组
float[] textures=new float[vCount*2];                    //顶点纹理坐标值数组
int count=0;                                             //顶点坐标数组的计数器
int stCount=0;                                           //顶点纹理坐标数组的计数器
for(float angdeg=0;Math.ceil(angdeg)<360;angdeg+=angdegSpan){//生成每对三角形的顶点数据
```

```java
double angrad=Math.toRadians(angdeg);                          //当前弧度
double angradNext=Math.toRadians(angdeg+angdegSpan);           //下一弧度
vertices[count++]=(float) (-r*Math.sin(angrad));               //第一三角形顶点 1
vertices[count++]=0;
    vertices[count++]=(float) (-r*Math.cos(angrad));
    textures[stCount++]=(float) (angrad/(2*Math.PI));          //纹理坐标
    textures[stCount++]=1;
    vertices[count++]=(float) (-r*Math.sin(angradNext));       //第一三角形顶点 2
    vertices[count++]=h;
    vertices[count++]=(float) (-r*Math.cos(angradNext));
    textures[stCount++]=(float) (angradNext/(2*Math.PI));      //纹理坐标
    textures[stCount++]=0;
    vertices[count++]=(float) (-r*Math.sin(angrad));           //第一三角形顶点 3
    vertices[count++]=h;
    vertices[count++]=(float) (-r*Math.cos(angrad));
    textures[stCount++]=(float) (angrad/(2*Math.PI));          //纹理坐标
    textures[stCount++]=0;
    vertices[count++]=(float) (-r*Math.sin(angrad));           //第二三角形顶点 1
    vertices[count++]=0;
    vertices[count++]=(float) (-r*Math.cos(angrad));
    textures[stCount++]=(float) (angrad/(2*Math.PI));          //纹理坐标
    textures[stCount++]=1;
    vertices[count++]=(float) (-r*Math.sin(angradNext));       //第二三角形顶点 2
    vertices[count++]=0;
    vertices[count++]=(float) (-r*Math.cos(angradNext));
    textures[stCount++]=(float) (angradNext/(2*Math.PI));      //纹理坐标
    textures[stCount++]=1;
    vertices[count++]=(float) (-r*Math.sin(angradNext));       //第二三角形顶点 3
    vertices[count++]=h;
    vertices[count++]=(float) (-r*Math.cos(angradNext));
    textures[stCount++]=(float) (angradNext/(2*Math.PI));      //纹理坐标
    textures[stCount++]=0;
  }
  float[] normals=new float[vertices.length];                  //法向量数据初始化
  for(int i=0;i<vertices.length;i++){
      if(i%3==1){normals[i]=0;                                 //法向量 Y 坐标分量为 0
      }else{normals[i]=vertices[i];}                           //法向量数组赋值
  }
```

说明

上述代码根据上一小节介绍的原理以一定的角度间距（angdegSpan）循环计算出了组成柱面的小三角形顶点的各项数据，包括顶点坐标、纹理坐标、法向量。要注意的是每次循环生成的是一对三角形顶点的相关数据，这是因为将柱面以一定的角度间距切分时，切分的结果是一个一个的四边形，每个四边形需要两个三角形构成。

（3）最后给出将圆面与柱面组装成圆柱体的相关代码，具体内容如下。

代码位置：见随书光盘中源代码/第8章/Sample8_1/com/bn/Sample8_1目录下的Cylinder.java。

```
//绘制顶面
MatrixState.pushMatrix();                                   //保护现场
```

```
MatrixState.translate(0, h/2*scale, 0);                //平移变换
MatrixState.rotate(-90, 1, 0, 0);                      //旋转变换
topCircle.drawSelf(topTexId);                          //绘制圆面
MatrixState.popMatrix();                               //恢复现场
//绘制底面
MatrixState.pushMatrix();                              //保护现场
MatrixState.translate(0, -h/2*scale, 0);               //平移变换
    MatrixState.rotate(90, 1, 0, 0);                   //旋转变换
    MatrixState.rotate(180, 0, 0, 1);                  //旋转变换
    bottomCircle.drawSelf(BottomTexId);                //绘制圆面
    MatrixState.popMatrix();                           //恢复现场
    //绘制侧面
    MatrixState.pushMatrix();                          //保护现场
    MatrixState.translate(0, -h/2*scale, 0);           //平移变换
    cylinderSide.drawSelf(sideTexId);                  //绘制柱面
    MatrixState.popMatrix();                           //恢复现场
```

提示

上述代码将圆面与柱面进行恰当的平移与旋转变换后组装成了圆柱体。另外，本案例中还有一套线段方式下生成顶点数据与组装成形的代码，基本套路与面填充方式的完全一致，这里不再赘述，需要的读者请参考随书光盘中的源代码。

8.2　圆锥体

上一节介绍了圆柱体的构建，本节将介绍圆锥体的构建。首先介绍圆锥面顶点原始位置生成的基本原理，然后给出实现的案例Sample8_2。

8.2.1　顶点原始位置的生成

本节中构建的圆锥体由两个部件组成，包括圆锥体底部的圆面和侧面的圆锥面。圆面顶点数据的计算在上一节中已经介绍过，因此这里仅介绍圆锥面顶点数据的计算。

1．顶点坐标的计算

圆锥面同样是由多个小三角形构成的，本节中以圆锥面顶端中心点为所有小三角形的共用顶点，再以一定的角度切分圆锥面底部的圆，得到各个小三角形另外两个顶点的坐标，具体情况如图8-7所示。

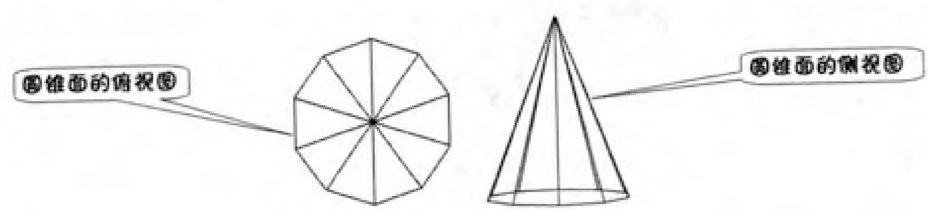


图8-7　圆锥侧面

> **提示**
>
> 从图8-7中看，构成的物体更像是棱锥而不是圆锥。这主要是因为切分的角度跨度较大，实际开发中选择合适的切分角度就可以解决这个问题。

从圆锥面的俯视图中可以看出，圆锥底边上顶点坐标的计算与圆面上顶点坐标计算的方法相同。设圆锥底面半径为R，圆锥底面中心位于坐标原点，圆锥中心轴与Y坐标轴重合，圆锥的高度为h，则圆锥底边某顶点的坐标计算公式如下。

$$P(x,y,z)=(R\cdot\sin\alpha,0,R\cdot\cos\alpha)$$

提示

上述公式中的a为顶点对应的切分角度，与图8-1中表示的含义相同。所不同的是图8-1中圆面位于XOY平面，这里是位于XOZ平面的。

另外，可以非常容易地直接得出圆锥面顶端中心点的坐标为P（*x,y,z*）=（0，h，0）。因此构成圆锥面的某一个小三角形3个顶点的坐标如下所列。

- *P*（*x,y,z*）=（*R*•sin *a*，0,*R*•cos *a*）
- *P*（*x,y,z*）=（0,h,0）
- *P*（*x,y,z*）=（*R*•sin（*a*+*span*），0,*R*•cos（*a*+*span*））。

提示

述顶点坐标公式中的span为切分的角度跨度，其越小则构建的图形越接近于圆锥。

2．顶点纹理坐标与法向量的计算

前面介绍完了顶点坐标数据的计算，下面将进一步介绍顶点纹理坐标与法向量的计算。由于圆锥底面的圆面中顶点纹理坐标与法向量的计算与上一节圆柱底面的完全相同，故这里不再赘述。另外，圆锥面顶点纹理坐标的计算与圆面也相同，因此这里仅介绍圆锥面中顶点法向量的计算，具体的思路如图8-8所示。

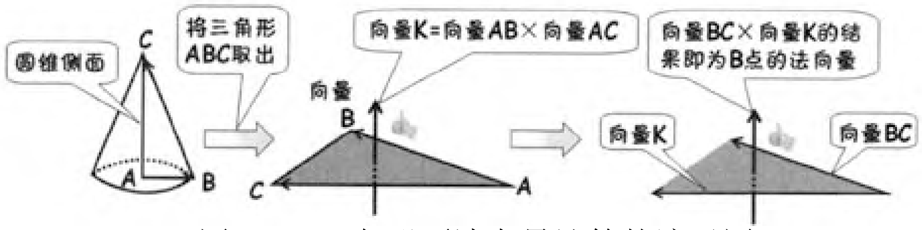


图8-8　三角形面法向量计算的演示图

提示

图8-8中A为圆锥底面中心点，B为圆锥底边圆周上的某点，C为圆锥顶部的点。

从图8-8中可以看出，圆锥面每一条棱上顶点的法向量都是相同的，因此对于每个具体切分角度对应的棱计算一次法向量即可。例如要计算BC棱上顶点的法向量，则需要首先计算出ABC面的法向量，再将此法向量与BC向量叉积即可得到BC棱上顶点的法向量。

ABC面法向量的计算也很简单，将向量AB与向量AC叉积即可得到。另外由于C点不仅属于一条棱，因此其法向量不能仅采用某一条棱的法向量，而应该采用所有棱法向量的平均值。根据圆锥的对称性可以容易地得出，此点法向量为[0，1，0]。

提示

本节法向量的计算用到了一些向量数学的相关知识，不熟悉的读者请参考其他数学资料，如《3D数学基础：图形与游戏开发》。

8.2.2　案例的开发

了解了圆锥体顶点坐标、纹理坐标、法向量的计算方法后，下面应该了解一下本节案例的运行效果了，如图8-9所示。

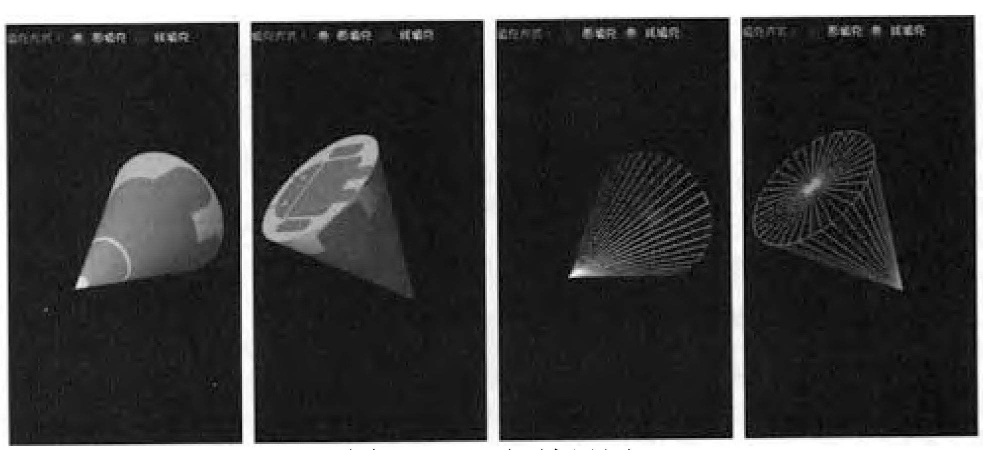


图8-9　运行效果图

说明

从图8-9中可以看出，本案例提供了两种绘制方式：面填充和线段。左侧两幅为面填充方式下本案例的运行效果图，右侧两幅为线段方式下本案例的运行效果图。

了解了本案例的运行效果后，就可以进行代码的开发了，这里仅给出本案例中有代表性的圆锥面中顶点坐标、纹理坐标、法向量生成的相关代码及向量计算工具类VectorUtil的代码，具体如下所列。

（1）首先给出的是圆锥面中顶点坐标、纹理坐标和法向量生成的相关代码，具体内容如下。

代码位置：见随书光盘中源代码/第8章/Sample8_2/com/bn/Sample8_2目录下的ConeSide.java。

```
float[] vertices=new float[vCount*3];            //顶点坐标数组
float[] textures=new float[vCount*2];            //顶点纹理坐标值数组
float[] normals=new float[vertices.length];      //法向量数组
int count=0;                                     //顶点坐标计数器
int stCount=0;                                   //纹理坐标计数器
int norCount=0;                                  //法向量计数器
```

```
for(float angdeg=0;Math.ceil(angdeg)<360;angdeg+=angdegSpan){//按照一定的角度跨度切分
    double angrad=Math.toRadians(angdeg);                              //当前弧度
    double angradNext=Math.toRadians(angdeg+angdegSpan);               //下一弧度
        vertices[count++]=0; vertices[count++]=h; vertices[count++]=0;
                                                                       //圆锥面中心最高点坐标
        textures[stCount++]=0.5f;textures[stCount++]=0;                //圆锥面中心最高点纹理坐标
        vertices[count++]=(float) (-r*Math.sin(angrad));               //当前弧度对应点坐标
        vertices[count++]=0;
        vertices[count++]=(float) (-r*Math.cos(angrad));
        textures[stCount++]=(float) (angrad/(2*Math.PI));              //当前弧度对应点纹理坐标
        textures[stCount++]=1;
        vertices[count++]=(float) (-r*Math.sin(angradNext));//下一弧度对应点坐标
        vertices[count++]=0;
        vertices[count++]=(float) (-r*Math.cos(angradNext));
        textures[stCount++]=(float) (angradNext/(2*Math.PI));//下一弧度对应点纹理坐标
        textures[stCount++]=1;
    }
    for(int i=0;i<vertices.length;i=i+3){                              //法向量数据的初始化
        if(vertices[i]==0&&vertices[i+1]==h&&vertices[i+2]==0){
                                                        //如果当前的顶点为圆锥的中心最高点
            normals[norCount++]=0;normals[norCount++]=1;normals[norCount++]=0;
        }else{                                                         //当前的顶点为底面圆周上的点
            float [] norXYZ=VectorUtil.calConeNormal(    //通过3个顶点的坐标求出法向量
                    0, 0, 0,                                           //底面圆的中心点
                    vertices[i], vertices[i+1], vertices[i+2],  //当前底面圆周顶点坐标
                    0, h, 0);                                          //圆锥中心最高点坐标
            normals[norCount++]=norXYZ[0];                             //将法向量X、Y、Z 3个分量的值
            normals[norCount++]=norXYZ[1];                             //存入法向量数据数组
            normals[norCount++]=norXYZ[2];
    }}
```

说明

上述代码根据上一小节介绍的原理以一定的角度间距（angdegSpan）循环计算出了组成圆锥面的各个小三角形顶点的各项数据，包括顶点坐标、纹理坐标、法向量等。

（2）接着给出的是用于计算圆锥面顶点法向量的工具类——VectorUtil，其代码如下。

代码位置：见随书光盘中源代码/第8章/Sample8_2/com/bn/Sample8_2目录下的VectorUtil.java。

```
package com.bn.Sample8_2;
public class VectorUtil {
    public static float[] calConeNormal(            //计算圆锥面指定棱顶点法向量的方法
            float x0,float y0,float z0,             //A，中心点(底面圆的圆心)
            float x1,float y1,float z1,             //B，底面圆上的某一点
            float x2,float y2,float z2 ){           //C，圆锥中心最高点
        float[] a={x1-x0, y1-y0, z1-z0};            //向量AB
        float[] b={x2-x0, y2-y0, z2-z0};            //向量AC
        float[] c={x2-x1, y2-y1, z2-z1};            //向量BC
            float[] k=crossTwoVectors(a,b);         //先求平面ABC的法向量k
            float[] d=crossTwoVectors(c,k);         //将c和k做叉乘，得出所求向量d
```

```
        return normalizeVector(d);              //返回规格化后的法向量
    }
    public static float[] normalizeVector(float [] vec){//向量规格化的方法
        float mod=module(vec);                  //求向量的模
        return new float[]{vec[0]/mod, vec[1]/mod, vec[2]/mod};//返回规格化后的向量
    }
    public static float module(float [] vec){        //求向量的模的方法
        return (float) Math.sqrt(vec[0]*vec[0]+vec[1]*vec[1]+vec[2]*vec[2]);
    }
    public static float[] crossTwoVectors( float[] a,float[] b){//求两个向量叉积的方法
        float x=a[1]*b[2]-a[2]*b[1];            //向量叉积的X分量
        float y=a[2]*b[0]-a[0]*b[2];            //向量叉积的Y分量
        float z=a[0]*b[1]-a[1]*b[0];            //向量叉积的Z分量
        return new float[]{x, y, z};            //返回叉积向量
}}
```

说明

该工具类中首先提供了向量计算的一些必备方法，如求模、规格化、求叉积等。另外还提供了专门为本案例服务的根据3个顶点的坐标计算法向量的方法calConeNormal，此方法实现了图8-8所示的计算思路。

8.3 圆环体

上一节介绍了圆椎体的构建，本节将介绍圆环体的构建。首先介绍圆环面顶点原始位置生成的基本原理，然后给出实现的案例Sample8_3。

8.3.1　顶点原始位置的生成

由于圆环体顶点法向量的计算较为复杂，因此本节案例中的圆环体仅采用了纹理映射，没有计算法向量并采用光照。为了计算的方便，本节案例中圆环面的生成方式为将XOY平面上的一个圆周绕*Y*轴旋转一周，具体情况如图8-10所示。

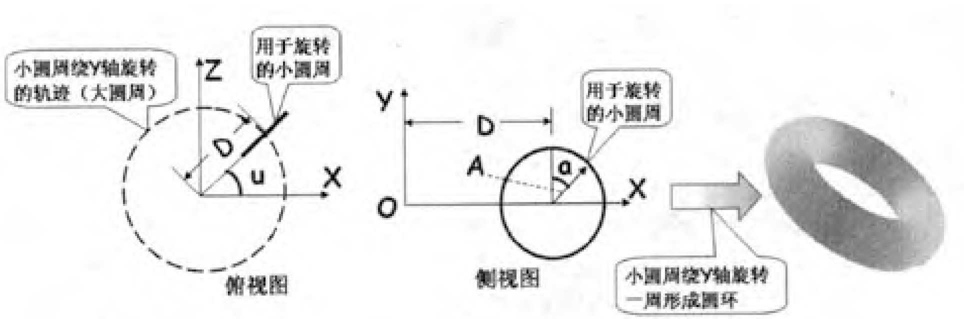


图8-10　圆环体的形成过程

假设该用于旋转的圆周半径为A，其圆心距*Y*轴的距离为D，且圆心位于XOZ平面上，同时此圆周绕*Y*轴旋转的轨迹形成的圆周也在XOZ平面上，则通过此方式形成的圆环面上任意顶点的坐标可以用如下公式表示。

$$P(x, y, z) = ((D + A \cdot \sin a)\sin u, A \cdot \cos a, (D + A \cdot \sin a)\cos u)$$

> **说明**
>
> 上述公式中a表示顶点在小圆周上对应的角度，u表示在大圆周上对应的角度。实际编程时可以用循环等间隔地依次取a与u的值即可得到构成圆环面各个顶点的坐标。

介绍完了顶点坐标的计算方法后，下面将简要介绍一下纹理坐标的生成思路，具体步骤如图8-11所示。

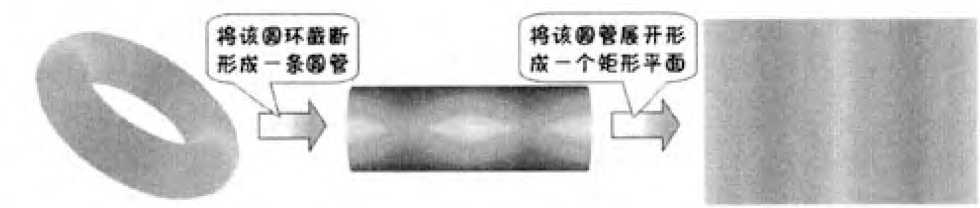


图8-11　圆环面展开成矩形

从图8-11中可以看出，最终纹理坐标的生成策略与前面圆柱面的没有本质区别，都是通过将目标曲面展开成矩形进行的。

提示

有的读者可能会有一个疑问：“为什么不介绍法向量的计算？难道复杂的物体就不需要光照吗？”请读者打消这个疑虑，后面的章节会介绍更为简便的复杂曲面物体呈现技术，当然包含光照方面的内容。这里略去是因为这种通过数学公式直接计算的方式对于复杂曲面而言工作量过大，很不合算。

8.3.2　案例的开发

了解了圆环体顶点坐标、纹理坐标、法向量的计算方法后，下面应该了解一下本节案例的运行效果了，如图8-12所示。

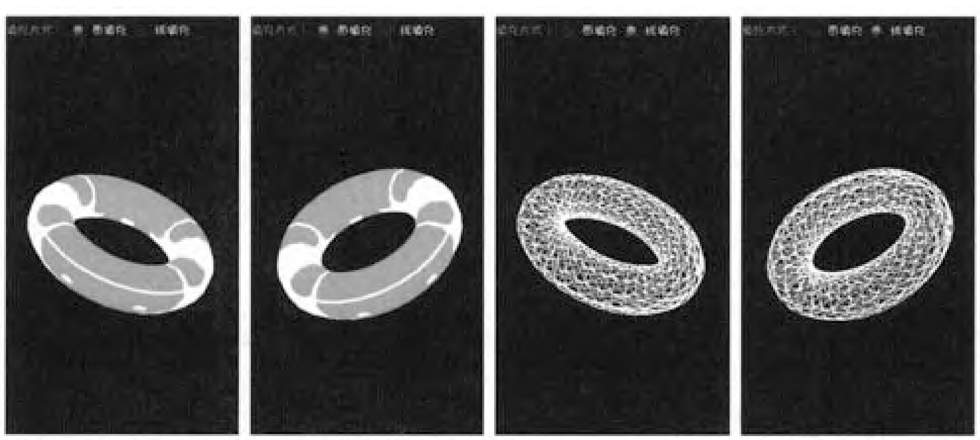


图8-12　运行效果图

说明

从图8-12中可以看出，本案例提供了两种绘制方式：面填充和线段。左侧两幅为面填充方式下本案例的运行效果图，右侧两幅为线段方式下本案例的运行效果图。

了解了本案例的运行效果后，就可以进行代码的开发了。这里仅给出本案例中有代表性的圆环顶点坐标、纹理坐标生成的相关代码，具体如下所列。

（1）首先给出的是圆环体顶点坐标、纹理坐标生成的相关代码，具体内容如下。

代码位置：见随书光盘中源代码/第8

章/Sample8_3/com/bn/Sample8_3目录下的Torus.java。

```
public void initVertexData(float rBig, float rSmall,         //圆环外径、内径
       int nCol ,int nRow) {                                  //小圆周和大圆周切分的份数
   float angdegColSpan=360.0f/nCol;                           //小圆周每份的角度跨度
   float angdegRowSpan=360.0f/nRow;                           //大圆周每份的角度跨度
   float A=(rBig-rSmall)/2;                                   //用于旋转的小圆半径
   float D=rSmall+A;                                          //旋转轨迹形成的大圆周半径
   vCount=3*nCol*nRow*2;                                      //顶点个数
   ArrayList<Float> alVertix=new ArrayList<Float>();//原始顶点列表（未卷绕）
   ArrayList<Integer> alFaceIndex=new ArrayList<Integer>();//用于组织三角形面的顶点编号列表
      for(float angdegCol=0;Math.ceil(angdegCol)<360+angdegColSpan;
      angdegCol+=angdegColSpan) {                             //对小圆按照等角度间距循环
         double a=Math.toRadians(angdegCol);                  //当前小圆弧度
         for(float angdegRow=0;Math.ceil(angdegRow)<360+angdegRowSpan;
            angdegRow+=angdegRowSpan) {                       //对大圆按照等角度间距循环
            double u=Math.toRadians(angdegRow);               //当前大圆弧度
            float y=(float) (A*Math.cos(a));                  //按照公式计算当前顶点
            float x=(float) ((D+A*Math.sin(a))*Math.sin(u));      //的 X、Y、Z 坐标
            float z=(float) ((D+A*Math.sin(a))*Math.cos(u));
            //将计算出来的 X、Y、Z 坐标放入原始顶点列表
            alVertix.add(x); alVertix.add(y); alVertix.add(z);
      }}
      for(int i=0;i<nCol;i++){                                //按照卷绕成三角形的需要
         for(int j=0;j<nRow;j++){                             //生成顶点编号列表
            int index=i*(nRow+1)+j;                           //当前四边形第一顶点编号
            alFaceIndex.add(index+1);         //第一个三角形三个顶点的编号入列表
            alFaceIndex.add(index+nRow+1);
            alFaceIndex.add(index+nRow+2);
            alFaceIndex.add(index+1);         //第二个三角形三个顶点的编号入列表
            alFaceIndex.add(index);
            alFaceIndex.add(index+nRow+1);
      }}
      float[] vertices=new float[vCount*3];//存放按照卷绕顺序顶点坐标值的数组
      cullVertex (alVertix, alFaceIndex, vertices);   //生成卷绕后的顶点坐标数组值
      ArrayList<Float> alST=new ArrayList<Float>();   //原纹理坐标列表（未卷绕）
      for(float angdegCol=0;Math.ceil(angdegCol)<360+angdegColSpan;
         angdegCol+=angdegColSpan){                           //对小圆按照等角度间距循环
         float t=angdegCol/360;                               //当前角度对应的 t 坐标
         for(float angdegRow=0;Math.ceil(angdegRow)<360+angdegRowSpan;
            angdegRow+=angdegRowSpan) {                       //对大圆按照等角度间距循环
```

```
            float s=angdegRow/360;                      //当前角度对应的s坐标
            alST.add(s); alST.add(t);                   //存入原始纹理坐标列表
        }}
        float[] textures= cullTexCoor (alST, alFaceIndex);  //生成卷绕后纹理坐标数组值
        ……//此处省略了将顶点坐标、纹理坐标数据存入缓冲的代码，与前面很多案例中的一致
}
```

说明

上述initVertexData方法根据上一小节介绍的公式以及切分的份数生成了渲染纹理圆环所需的顶点坐标数据与纹理坐标数据。

（2）为了减少重复代码，本案例中将按照顶点编号生成卷绕后顶点坐标数组及纹理坐标数组的工作独立成了两个方法：cullVertex与cullTexCoor，具体代码如下。

代码位置：见随书光盘中源代码/第8章/Sample8_3/com/bn/Sample8_3目录下的Torus.java。

```
public static void cullVertex (                       //根据顶点编号生成卷绕后顶点坐标数组的方法
        ArrayList<Float> alv,                         //原始顶点列表
        ArrayList<Integer> alFaceIndex,               //用于组织三角形面的顶点编号列表
        float[] vertices){                            //存放卷绕后顶点坐标值的数组
    int vCount=0;                                     //顶点计数器
    for(int i:alFaceIndex){                           //对顶点编号列表进行循环
        vertices[vCount++]=alv.get(3*i);              //将当前编号顶点的 X 坐标值存入最终数组
        vertices[vCount++]=alv.get(3*i+1);            //将当前编号顶点的 Y 坐标值存入最终数组
        vertices[vCount++]=alv.get(3*i+2);            //将当前编号顶点的 Z 坐标值存入最终数组
    }}
    public static float[] cullTexCoor (               //根据顶点编号生成卷绕后顶点纹理坐标数组的方法
            ArrayList<Float> alST,                    //原始纹理坐标列表
            ArrayList<Integer> alTexIndex){           //用于组织三角形面的顶点编号列表
        float[] textures=new float[alTexIndex.size()*2];   //结果纹理坐标数组
        int stCount=0;                                //纹理坐标计数器
        for(int i:alTexIndex){                        //对顶点编号列表进行循环
            textures[stCount++]=alST.get(2*i);        //将当前编号顶点的 S 坐标值存入最终数组
            textures[stCount++]=alST.get(2*i+1);      //将当前编号顶点的 T 坐标值存入最终数组
        }
        return textures;                              //返回结果纹理坐标数组
    }
```

说明

上述两个方法的代码非常简单，都是按照用于组织三角形面的顶点编号列表中的顶点编号将卷绕后的顶点坐标或纹理坐标数据放进对应数组并返回。

8.4 螺旋管

上一节介绍了圆环体的构建，本节将介绍螺旋管的构建。首先介绍螺旋管面顶点原始位置生成的基本原理，然后给出实现的案例Sample8_4。

8.4.1 顶点原始位置的生成

通过上一节的学习掌握了圆环体的构建后，以此为基础再构建螺旋管就非常容易了。螺旋管可以看作在某处切开并沿中心轴拉开的圆环，具体情况如图8-13所示。

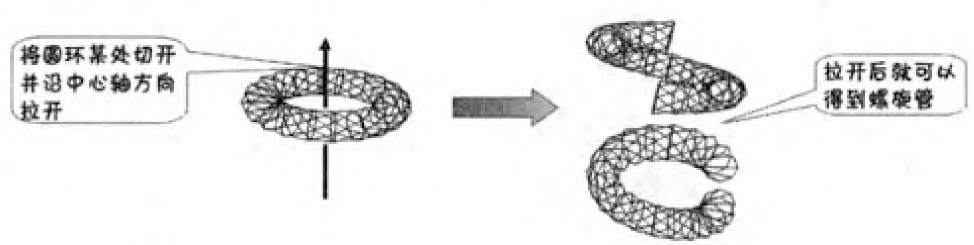


图8-13　基于圆环生成螺旋管

图8-13中得到的螺旋管只有一周，若希望得到绕中心轴多周的类似弹簧的螺旋管只需要让拉开前圆环不止绕中心轴一周即可，如图8-14所示。

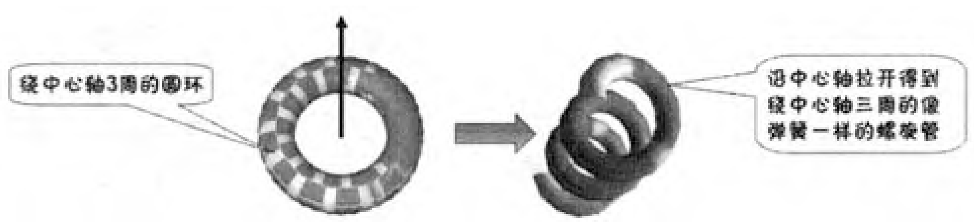


图8-14　绕中心轴3周的圆环拉开

提示

实际开发时螺旋管顶点坐标的生成与圆环顶点坐标的生成代码基本相同，唯一不同的是Y坐标不再是固定的，而是根据当前绕中心轴的旋转角度换算得出。也就是小圆周一边绕中心轴旋转一边变

换Y坐标。而纹理坐标的生成就几乎没有区别了，都是最后展开成矩形与纹理图对应。

8.4.2 案例的开发

了解了螺旋管顶点坐标、纹理坐标的计算方法后，下面应该了解一下本节案例的运行效果了，如图8-15所示。

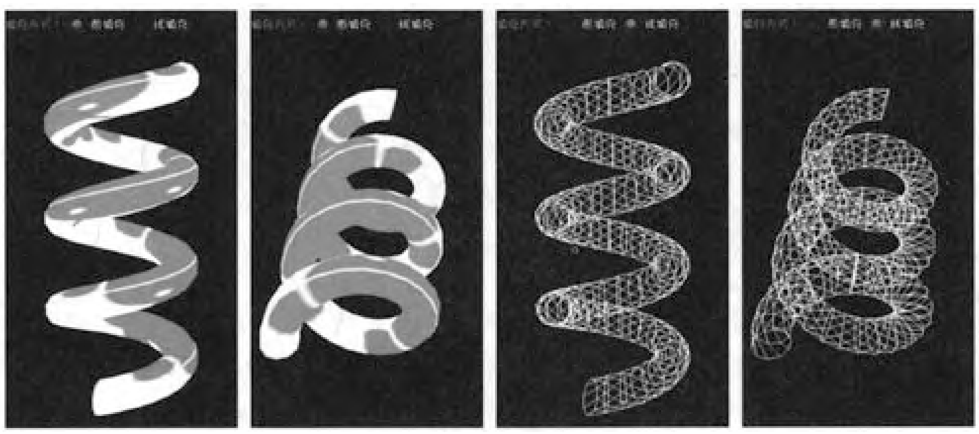


图8-15 运行效果图

说明

从图8-15中可以看出，本案例提供了两种绘制方式：面填充和线段。左侧两幅为面填充方式下本案例的运行效果图，右侧两幅为线段方式下本案例的运行效果图。

了解了本案例的运行效果后，就可以进行代码的开发了，这里仅给出本案例中有代表性的螺旋管顶点坐标、纹理坐标生成的相关代码，具体如下所列。

代码位置：见随书光盘中源代码/第8章/Sample8_4/com/bn/Sample8_4目录下的Spring.java。

```
public void initVertexData(float rBig, float rSmall, //螺旋管外径、螺旋管内径
float h, float nCirclef,                              //螺旋管高度，螺旋管圈数
int nCol ,int nRow) {                                 //小圆周和大圆周切分的份数
 float angdegTotal=nCirclef*360.0f;                   //大圆周总度数
 float angdegColSpan=360.0f/nCol;                     //小圆周每份的角度跨度
 float angdegRowSpan=angdegTotal/nRow;                //大圆周每份的角度跨度
 float A=(rBig-rSmall)/2;                             //用于旋转的小圆半径
 float D=rSmall+A;                                    //旋转轨迹形成的大圆周半径
 vCount=3*nCol*nRow*2;                                //顶点个数
     ArrayList<Float> alVertix=new ArrayList<Float>(); //原始顶点列表（未卷绕）
     ArrayList<Integer> alFaceIndex=new ArrayList<Integer>();
                                                      //用于组织三角形面的顶点编号列表
     for(float angdegCol=0;Math.ceil(angdegCol)<360+angdegColSpan;
     angdegCol+=angdegColSpan){                       //对小圆按照等角度间距循环
       double a=Math.toRadians(angdegCol);            //当前小圆周弧度
       for(float angdegRow=0;Math.ceil(angdegRow)<angdegTotal+angdegRowSpan;
       angdegRow+=angdegRowSpan){                     //对大圆按照等角度间距循环
         float yVec=(angdegRow/angdegTotal)*h  ;//根据大圆周旋转角度折算出小圆中心 Y 坐标
         double u=Math.toRadians(angdegRow);                  //当前大圆周弧度
         float y=(float) (A*Math.cos(a));                     //按照公式计算当前顶点
         float x=(float) ((D+A*Math.sin(a))*Math.sin(u));     //的 X、Y、Z 坐标
         float z=(float) ((D+A*Math.sin(a))*Math.cos(u));
         //将计算出来的 X、Y、Z 坐标放入原始顶点列表
         alVertix.add(x); alVertix.add(y+yVec); alVertix.add(z);
     }}
     ……//由于该部分代码与圆环案例顶点数据初始化的代码相同，此处不再进行重复介绍
   }
```

说明

从上述代码中可以看出，除了绕中心轴旋转的圈数不再是一圈、小圆周在绕中心轴旋转过程中其圆心*Y*坐标不断根据绕中心轴旋转的角度变化外，螺旋管顶点坐标、纹理坐标生成的代码与圆环完全相同。

8.5　几何球

上一节介绍了螺旋管的构建，本节将介绍几何球的构建。首先介绍几何球顶点原始位置生成的基本原理，然后给出实现的案例Sample8_5。

8.5.1　顶点原始位置的生成

前面早在第7章地月系场景的案例中就介绍过球体的构建，当时采用的是按照经纬度进行切分的方式，构建出的球体视觉效果很不错。但如果将普通的花纹纹理映射到这种方式生成的球面上，就会发现南北两极的纹理被压缩，赤道位置的纹理被拉申，变形情况比较严重，如图8-16所示。

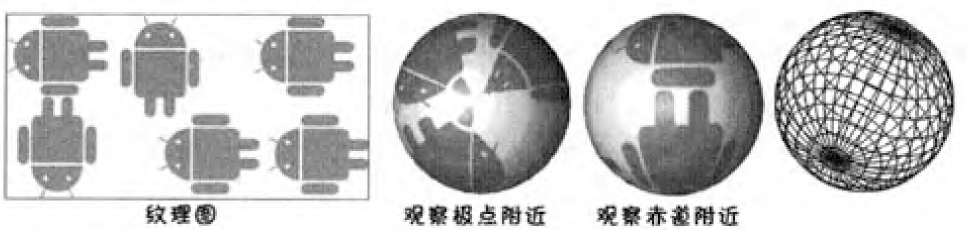


图8-16　经纬度切分的球面纹理变形情况

提示

读者可能会有这样的疑问：“图8-16中的纹理图映射到球面后是变形了，但地月系场景中并没有这种感觉啊？”这是因为地月系

场景中采用的纹理图本来就是变形的，经过纹理映射再次变形后反而恢复了正常。

造成如图8-16中变形的原因是越靠近两极顶点越密集，越靠近赤道顶点越稀疏。而计算纹理坐标时并没有考虑这一点，认为顶点是均匀分布的。因此，若希望在应用纹理时没有上述变形的情况就不能采用此种方式构建的球体了，而应该采用本节介绍的几何球来实现。

本节介绍的几何球是在正二十面体的基础上拓展形成的，基本思路为对正二十面体每条边所对应的弧进行切分，从而得到所需顶点数量对应的几何球体。下面分两方面对几何球顶点数据的生成进行介绍，包括顶点坐标和法向量的计算与顶点纹理坐标的计算。

1．顶点坐标和法向量的计算

由于本节中的几何球是从正二十面体拓展而来，因此应该首先了解一下正二十面体各个顶点的生成方式，如图8-17所示。

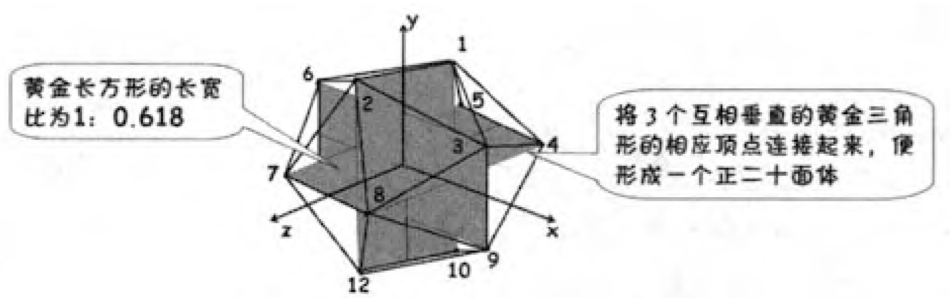


图8-17　用黄金长方形定义正二十面体

从图8-17中可以看出，正二十面体的12个顶点可以通过3个分别位于XOY、XOZ、YOZ平面并正交（所谓正交就是互相垂直）于坐标原点的矩形生成。若希望生成的二十面体是正二十面体，则对矩形的长宽比有严格的要求。具体说就是长宽比为1：0.618，也就是黄金分割比。

得到了正二十面体后，再对正二十面体每条边所对应的外接圆上的弧进一步切分，即可形成几何球，如图8-18所示。

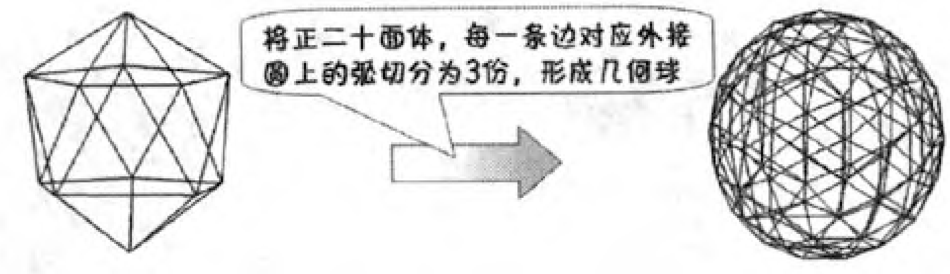


图8-18　几何球示意图

提示

随着正二十面体每条边对应外接圆上的弧切分份数的增加，最终形成的几何球表面将越来越平滑。而球面上每个顶点的坐标值，可以通过弧切分的份数、球的半径、弧对应正二十面体边的顶点，并根据三角函数求出。

从图8-18中可以看出，几何球上顶点的一个重要特性是：任何顶点没有位置的特殊性，都是对等的。另外，由于求几何球原始顶点坐标时球心是位于坐标原点的，因此每个顶点的坐标值同时也可以作为法向量使用。

2．顶点纹理坐标的计算

了解了几何球顶点坐标与法向量的生成方式后，下面介绍几何球上顶点纹理坐标的计算。同样还是要从正二十面体开始，首先将正二十面体展开，如图8-19所示。

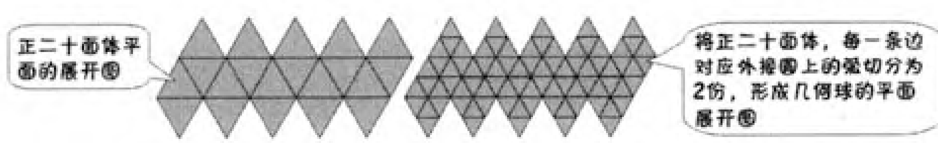


图8-19　正二十面体和几何球的平面展开图

图8-19中左侧为正二十面体展开后的平面图，右侧为将正二十面体每条边对应圆弧切分为两份后形成的几何球展开后的平面图。因此

可以想到，计算纹理坐标时只要将展开后的平面对应到二维纹理图上并计算出每个顶点的ST纹理坐标即可，如图8-20所示。

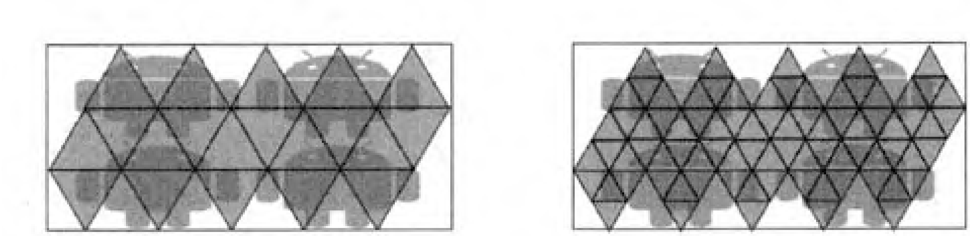


图8-20　几何球顶点纹理坐标的生成

提示

从图8-20中可以想象出，由于所有的顶点在纹理图上是均匀分布的，处处密度相同，因此不再会有按照经纬度进行切分生成的球面纹理变形的问题。但几何球的纹理映射也有一个明显的缺点，那就是不能将纹理图中的所有内容都映射到球面上，因此在组织纹理图内容时最好注意这一点，尽量将纹理有效内容调整到能映射的区域。

8.5.2　案例的开发

了解了几何球顶点坐标、纹理坐标、法向量的计算方法后，下面应该了解一下本节案例的运行效果了，如图8-21所示。

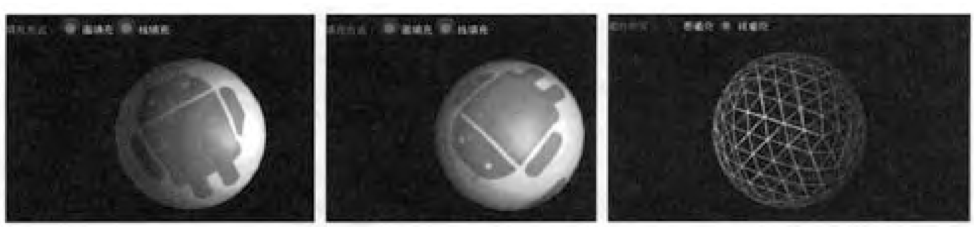


图8-21　几何球案例运行效果图

说明

从图8-21中可以看出，本案例提供了两种绘制方式：面填充和线段。左侧两幅为面填充方式下本案例的运行效果图，右侧一幅为

线段方式下本案例的运行效果图。另外可以看出，纹理映射到几何球球面上后没有产生原来球体的那种纹理变形。

了解了本案例的运行效果后，就可以进行代码的开发了。这里仅给出本案例中有代表性的几何球顶点坐标、纹理坐标、法向量生成的相关代码，具体如下所列。

（1）首先给出的是用于初始化几何球顶点数据的initVertexData方法，其代码如下。

代码位置：见随书光盘中源代码/第8章/Sample8_5/com/bn/Sample8_5目录下的Regular20.java。

```
public void initVertexData(float scale, float aHalf, int n){ //自定义的初始化顶点数据的方法
    aHalf*=scale;                                               //黄金长方形长边的一半
    bHalf=aHalf*0.618034f;                                      //黄金长方形短边的一半
    r=(float) Math.sqrt(aHalf*aHalf+bHalf*bHalf);               //几何球的半径
    vCount=3*20*n*n;                                            //顶点个数
    ArrayList<Float> alVertix20=new ArrayList<Float>();         //正二十面体的顶点列表
    //用于卷绕构成正二十面体各个三角形的顶点编号列表
    ArrayList<Integer> alFaceIndex20=new ArrayList<Integer>();
    initAlVertix20(alVertix20,aHalf,bHalf);             //初始化正二十面体的顶点坐标数据
        initAlFaceIndex20(alFaceIndex20);//初始化用于卷绕构成正二十面体各个三角形的顶点编号列表
        //构成正二十面体的各个三角形顶点的坐标数据数组
        float[] vertices20=Vector3Util.cullVertex (alVertix20, alFaceIndex20);
        ArrayList<Float> alVertix=new ArrayList<Float>();          //几何球原始顶点列表
        ArrayList<Integer> alFaceIndex=new ArrayList<Integer>();
                                                        //构成几何球的各三角形顶点编号列表
        int vnCount=0;                                  //顶点计数器
        for(int k=0;k<vertices20.length;k+=9){          //对正二十面体中的每个三角形循环
          float [] v1=new float[]{vertices20[k+0], vertices20[k+1], vertices20[k+2]};
                                                        //当前三角形 3 个
          float [] v2=new float[]{vertices20[k+3], vertices20[k+4], vertices20[k+5]};
                                                        //顶点的坐标
          float [] v3=new float[]{vertices20[k+6], vertices20[k+7], vertices20[k+8]};
          for(int i=0;i<=n;i++){                  //根据切分的份数求出几何球原始顶点的坐标
              float[] viStart=Vector3Util.devideBall(r, v1, v2, n, i);//对圆弧进行切分
              float[] viEnd=Vector3Util.devideBall(r, v1, v3, n, i); //对圆弧进行切分
              for(int j=0;j<=i;j++){
               float[] vi=Vector3Util.devideBall(r, viStart, viEnd, i, j);
```

```
                                        //对圆弧进行切分
            alVertix.add(vi[0]); alVertix.add(vi[1]); alVertix.add(vi[2]);
                                        //将坐标存入原始顶点列表
         }}
         for(int i=0;i<n;i++){          //循环生成构成几个球各个三角形的顶点编号列表
            if(i==0){                   //若是第0行，顶点编号012
             alFaceIndex.add(vnCount+0);alFaceIndex.add(vnCount+1);alFaceIndex.
             add(vnCount+2);
             vnCount+=1;                //顶点计数器加1
             //如果是正二十面体三角形的最后一次循环，将下一拨的顶点个数也加上
             if(i==n-1){ vnCount+=2; }
                  continue;
            }
            int iStart=vnCount;      //第i行开始的编号(这里的行指的是平面展开图中的行)
            int viCount=i+1;                            //第i行顶点数
            int iEnd=iStart+viCount-1;                  //第i行结束顶点编号
            int iStartNext=iStart+viCount;              //第i+1行开始的顶点编号
            int viCountNext=viCount+1;                  //第i+1行顶点数
            int iEndNext=iStartNext+viCountNext-1;      //第i+1行结束的顶点编号
            for(int j=0;j<viCount-1;j++){               //前面的四边形
               int index0=iStart+j;                     //四边形4个顶点的编号
               int index1=index0+1;
               int index2=iStartNext+j;
               int index3=index2+1;
               //将四边形4个顶点卷绕成两个三角形
               alFaceIndex.add(index0); alFaceIndex.add(index2);alFaceIndex.add
               (index3);
               alFaceIndex.add(index0); alFaceIndex.add(index3);alFaceIndex.add
               (index1);
            }
            //最后一个三角形3个顶点的编号
            alFaceIndex.add(iEnd); alFaceIndex.add(iEndNext-1);alFaceIndex.add
           (iEndNext);
            vnCount+=viCount;                           //第i行前所有顶点数的和
          ' if(i==n-1){ //如果是正二十面体三角形的最后一次循环，将下一拨的顶点个数也加上
               vnCount+=viCountNext;
         }}}
      //按照生成的顶点编号序列填充顶点坐标数据数组
      float[] vertices=Vector3Util.cullVertex (alVertix, alFaceIndex);
      float[] normals=vertices;                         //顶点坐标就是法向量
      ArrayList<Float> alST20=new ArrayList<Float>(); //正二十面体顶点的原始纹理坐标列表
      ArrayList<Integer> alTexIndex20=new ArrayList<Integer>();
                                                        //正二十面体纹理坐标编号列表
      float sSpan=1/5.5f;                               //每个纹理三角形的边长
      float tSpan=1/3.0f;                               //每个纹理三角形的高
      //下面4个循环生成了正二十面体展开后各个顶点的纹理坐标
      for(int i=0;i<5;i++){ alST20.add(sSpan+sSpan*i); alST20.add(0f); }
      for(int i=0;i<6;i++){ alST20.add(sSpan/2+sSpan*i); alST20.add(tSpan); }
      for(int i=0;i<6;i++){ alST20.add(sSpan*i); alST20.add(tSpan*2);  }
      for(int i=0;i<5;i++){ alST20.add(sSpan/2+sSpan*i); alST20.add(tSpan*3);   }
```

```
        initAlTexIndex20(alTexIndex20); //生成组成正二十面体各个三角形顶点的纹理坐标编号列表
        //按照生成的顶点编号序列填充顶点纹理坐标数据数组(正二十面体的)
        float[] st20=Vector3Util.cullTexCoor(alST20, alTexIndex20);
        ArrayList<Float> alST=new ArrayList<Float>(); //几何球的原始纹理坐标列表(未卷绕)
        for(int k=0;k<st20.length;k+=6){            //对正二十面体的各个三角形进行循环
            float [] st1=new float[]{st20[k+0], st20[k+1], 0};//取出当前三角形
            float [] st2=new float[]{st20[k+2], st20[k+3], 0};//3个顶点的纹理坐标
            float [] st3=new float[]{st20[k+4], st20[k+5], 0};
            for(int i=0;i<=n;i++){                  //对正二十面体平面展开图的边进行切分
                float[] stiStart=Vector3Util.devideLine(st1, st2, n, i);
                float[] stiEnd=Vector3Util.devideLine(st1, st3, n, i);
                for(int j=0;j<=i;j++){          //计算几何球平面展开图顶点对应的纹理坐标
                    float[] sti=Vector3Util.devideLine(stiStart, stiEnd, i, j);
                    alST.add(sti[0]); alST.add(sti[1]);    //将纹理坐标加入列表
        }}}
        //按照生成的顶点编号序列填充顶点纹理坐标数据数组(几何球的)
        float[] textures=Vector3Util.cullTexCoor(alST, alFaceIndex); //计算卷绕后纹理坐标
        ......//此处省略了生成顶点坐标、顶点纹理坐标、顶点法向量数据缓冲的代码
}
```

说明

上述代码实现了上一小节中介绍的从二十面体生成几何球各个顶点坐标及纹理坐标的方法，细节比较多，读者若想彻底掌握需要花时间细细研读代码。若读者仅仅是需要使用几何球则可以直接复制光盘中的源代码来使用。

（2）接着给出的是初始化正二十面体顶点坐标数据的initAlVertix20方法，其代码如下。

代码位置：见随书光盘中源代码/第8章/Sample8_5/com/bn/Sample8_5目录下的Regular20.java。

```
public void initAlVertix20(ArrayList<Float> alVertix20,float aHalf,float bHalf){
    alVertix20.add(0f); alVertix20.add(aHalf); alVertix20.add(-bHalf);
                                                        //对应图 8-17 的 1 号点
    alVertix20.add(0f); alVertix20.add(aHalf); alVertix20.add(bHalf); //对应图 8-17 的 2 号点
    alVertix20.add(aHalf); alVertix20.add(bHalf); alVertix20.add(0f); //对应图 8-17 的 3 号点
    alVertix20.add(bHalf); alVertix20.add(0f); alVertix20.add(-aHalf);
                                                        //对应图 8-17 的 4 号点
    alVertix20.add(-bHalf); alVertix20.add(0f); alVertix20.add(-aHalf);
                                                        //对应图 8-17 的 5 号点
    alVertix20.add(-aHalf); alVertix20.add(bHalf); alVertix20.add(0f);
                                                        //对应图 8-17 的 6 号点
    alVertix20.add(-bHalf); alVertix20.add(0f); alVertix20.add(aHalf);//对应图 8-17 的 7 号点
    alVertix20.add(bHalf); alVertix20.add(0f); alVertix20.add(aHalf);//对应图 8-17 的 8 号点
        alVertix20.add(aHalf); alVertix20.add(-bHalf); alVertix20.add(0f);
                                                        //对应图 8-17 的 9 号点
        alVertix20.add(0f); alVertix20.add(-aHalf); alVertix20.add(-bHalf);
                                                        //对应图 8-17 的 10 号点
        alVertix20.add(-aHalf); alVertix20.add(-bHalf); alVertix20.add(0f);
                                                        //对应图 8-17 的 11 号点
        alVertix20.add(0f); alVertix20.add(-aHalf); alVertix20.add(bHalf);
                                                        //对应图 8-17 的 12 号点
}
```

说明

该方法实现了图8-17中给出的正二十面体顶点坐标生成方法，其中aHalf和bHalf为黄金长方形长边和短边的半边长。想彻底掌握上述方法的读者请在阅读代码的同时细致参照图8-17，仅阅读代码是没有意义的。

（3）下面给出的是初始化构成正二十面体的20个三角形顶点坐标编号数据的initAlFaceIndex20方法，其代码如下。

代码位置：见随书光盘中源代码/第8章/Sample8_5/com/bn/Sample8_5目录下的Regular20.java。

```java
public void initAlFaceIndex20(ArrayList<Integer> alFaceIndex20){
    alFaceIndex20.add(0); alFaceIndex20.add(1); alFaceIndex20.add(2);//第一行5个三角形
    alFaceIndex20.add(0); alFaceIndex20.add(2); alFaceIndex20.add(3);//的各个顶点的坐标
    alFaceIndex20.add(0); alFaceIndex20.add(3); alFaceIndex20.add(4);//编号
    alFaceIndex20.add(0); alFaceIndex20.add(4); alFaceIndex20.add(5);
    alFaceIndex20.add(0); alFaceIndex20.add(5); alFaceIndex20.add(1);
    alFaceIndex20.add(1); alFaceIndex20.add(6); alFaceIndex20.add(7);//第二行10个三角形
    alFaceIndex20.add(1); alFaceIndex20.add(7); alFaceIndex20.add(2);//的各个顶点的坐标
    alFaceIndex20.add(2); alFaceIndex20.add(7); alFaceIndex20.add(8);  //编号
        alFaceIndex20.add(2); alFaceIndex20.add(8); alFaceIndex20.add(3);
        alFaceIndex20.add(3); alFaceIndex20.add(8); alFaceIndex20.add(9);
        alFaceIndex20.add(3); alFaceIndex20.add(9); alFaceIndex20.add(4);
        alFaceIndex20.add(4); alFaceIndex20.add(9); alFaceIndex20.add(10);
        alFaceIndex20.add(4); alFaceIndex20.add(10); alFaceIndex20.add(5);
        alFaceIndex20.add(5); alFaceIndex20.add(10); alFaceIndex20.add(6);
        alFaceIndex20.add(5); alFaceIndex20.add(6); alFaceIndex20.add(1);
        alFaceIndex20.add(6); alFaceIndex20.add(11); alFaceIndex20.add(7);
                                                            //第三行5个三角形
        alFaceIndex20.add(7); alFaceIndex20.add(11); alFaceIndex20.add(8);
                                                            //的各个顶点的坐标
        alFaceIndex20.add(8); alFaceIndex20.add(11); alFaceIndex20.add(9); //编号
        alFaceIndex20.add(9); alFaceIndex20.add(11); alFaceIndex20.add(10);
        alFaceIndex20.add(10); alFaceIndex20.add(11); alFaceIndex20.add(6);
    }
```

说明

该方法根据图8-17及图8-19给出了组成正二十面体的各个三角形3个顶点的编号序列，想彻底掌握上述方法的读者请在阅读代码的

同时细致参照图8-17与图8-19，仅阅读代码是没有意义的。

（4）接着给出的是初始化构成正二十面体的20个三角形各个顶点纹理坐标编号数据的initA1TexIndex20方法，具体内容如下。

代码位置：见随书光盘中源代码/第8章/Sample8_5/com/bn/Sample8_5目录下的Regular20.java。

```
public void initAlTexIndex20(ArrayList<Integer> alTexIndex20){
    alTexIndex20.add(0); alTexIndex20.add(5); alTexIndex20.add(6);//第一行5个三角形
    alTexIndex20.add(1); alTexIndex20.add(6); alTexIndex20.add(7);//的各个顶点的纹理
    alTexIndex20.add(2); alTexIndex20.add(7); alTexIndex20.add(8);//坐标编号
    alTexIndex20.add(3); alTexIndex20.add(8); alTexIndex20.add(9);
```

```java
alTexIndex20.add(4); alTexIndex20.add(9); alTexIndex20.add(10);
alTexIndex20.add(5); alTexIndex20.add(11); alTexIndex20.add(12);//第二行10个三角形
alTexIndex20.add(5); alTexIndex20.add(12); alTexIndex20.add(6);//的各个顶点的纹理
alTexIndex20.add(6); alTexIndex20.add(12); alTexIndex20.add(13); //坐标编号
    alTexIndex20.add(6); alTexIndex20.add(13); alTexIndex20.add(7);
    alTexIndex20.add(7); alTexIndex20.add(13); alTexIndex20.add(14);
    alTexIndex20.add(7); alTexIndex20.add(14); alTexIndex20.add(8);
    alTexIndex20.add(8); alTexIndex20.add(14); alTexIndex20.add(15);
    alTexIndex20.add(8); alTexIndex20.add(15); alTexIndex20.add(9);
    alTexIndex20.add(9); alTexIndex20.add(15); alTexIndex20.add(16);
    alTexIndex20.add(9); alTexIndex20.add(16); alTexIndex20.add(10);
    alTexIndex20.add(11); alTexIndex20.add(17); alTexIndex20.add(12);
                                                        //第三行5个三角形
    alTexIndex20.add(12); alTexIndex20.add(18); alTexIndex20.add(13);
                                                        //的各个顶点的纹理
    alTexIndex20.add(13); alTexIndex20.add(19); alTexIndex20.add(14); //坐标编号
    alTexIndex20.add(14); alTexIndex20.add(20); alTexIndex20.add(15);
    alTexIndex20.add(15); alTexIndex20.add(21); alTexIndex20.add(16);
}
```

说明

上述代码是根据图8-20开发的，请读者在细致阅读代码的同时参考图8-20，仅直接阅读代码很难理解。

（5）最后给出的是向量计算及圆弧拆分相关方法所属的工具类——VectorUtil，其代码如下。

代码位置：见随书光盘中源代码/第8章/Sample8_5/com/bn/Sample8_5目录下的VectorUtil.java。

```
package com.bn.Sample8_5;                                    //包声明
import java.util.ArrayList;                                  //相关类的引入
public class Vector3Util {
    ......//此处省略向量规格化的 normalizeVector 方法
    ......//此处省略向量求模的 module 方法
    ......//此处省略向量叉积的 crossTwoVectors 方法
    public static float dotTwoVectors(float[] a, float[] b){ //两个向量点积的方法
        return a[0]*b[0]+a[1]*b[1]+a[2]*b[2];                //返回点积结果
    }
        //根据顶点编号生成卷绕后顶点纹理坐标数组的方法
        public static float[] cullTexCoor(ArrayList<Float> alST,ArrayList<Integer>
        alTexIndex){
            float[] textures=new float[alTexIndex.size()*2];//结果纹理坐标数组
            int stCount=0;                                   //纹理坐标计数器
            for(int i:alTexIndex){                           //对顶点编号列表进行循环
                textures[stCount++]=alST.get(2*i); //将当前编号顶点的S坐标值存入最终数组
                textures[stCount++]=alST.get(2*i+1);//将当前编号顶点的T坐标值存入最终数组
            }
            return textures;                        //返回结果纹理坐标数组
        }
        //根据顶点编号生成卷绕后顶点坐标数组的方法
        public static float[] cullVertex(ArrayList<Float> alv,ArrayList<Integer>
        alFaceIndex){
```

```
        float[] vertices=new float[alFaceIndex.size()*3];//存放卷绕后顶点坐标值的数组
        int vCount=0;                                   //顶点计数器
        for(int i:alFaceIndex){                         //对顶点编号列表进行循环
            vertices[vCount++]=alv.get(3*i);    //将当前编号顶点的 X 坐标值存入最终数组
            vertices[vCount++]=alv.get(3*i+1);  //将当前编号顶点的 Y 坐标值存入最终数组
            vertices[vCount++]=alv.get(3*i+2);  //将当前编号顶点的 Z 坐标值存入最终数组
        }
        return vertices;                                //返回结果顶点坐标数组
    }
    //计算圆弧的 n 等分点坐标的方法，r 为半径，start 为从球心到圆弧起始点的向量，
    //end 为从球心到圆弧终点的向量，n 为切分的总份数，i 为所求点对应的份数编号
    public static float[] devideBall(float r,float[] start,float[] end,int n,int i){
        float[] s=Vector3Util.normalizeVector(start);//先将指向起点和终点的向量规格化
        float[] e=Vector3Util.normalizeVector(end);
        if(n==0){ return new float[]{s[0]*r, s[1]*r, s[2]*r};}
                                                        //如果 n 为零，返回起点坐标
        double angrad=Math.acos(Vector3Util.dotTwoVectors(s, e));//起点终点向量夹角
        double angrad1=angrad*i/n;                      //所求向量和起点向量的夹角
        double angrad2=angrad-angrad1;                  //所求向量和终点向量的夹角
        float[] normal=Vector3Util.crossTwoVectors(s, e);//求与 s、e 向量正交的向量
        double matrix[][]={//用 doolittle 分解算法解 n 元一次线性方程组所需的系数矩阵
                {s[0],s[1],s[2],Math.cos(angrad1)},
                {e[0],e[1],e[2],Math.cos(angrad2)},
                {normal[0],normal[1],normal[2],0}
        };
        double result[]=MyMathUtil.doolittle(matrix);    //解 n 元一次线性方程组
        float x=(float) result[0];                      //得到从球心到所求点向量的规格化版本
        float y=(float) result[1];
        float z=(float) result[2];
        return new float[]{x*r, y*r, z*r};              //得到所求点的坐标并返回
    }
    //计算线段的 n 等分点坐标的方法，start 线段起点坐标，end 为线段终点坐标
    //n 为切分的总份数，i 为所求点对应的份数编号
    public static float[] devideLine(float[] start,float[] end,int n, int i ){
        if(n==0){ return start;   }                     //如果 n 为零，返回起点坐标
        //求起点到终点的向量
        float[] ab=new float[]{end[0]-start[0], end[1]-start[1], end[2]-start[2]};
        float vecRatio=i/(float)n;                      //求向量比例
        //求起点到所求点的向量
        float[] ac=new float[]{ab[0]*vecRatio, ab[1]*vecRatio, ab[2]*vecRatio};
        float x=start[0]+ac[0]; float y=start[1]+ac[1]; float z=start[2]+ac[2];
                                                        //得到所求点坐标
        return new float[]{x, y, z};                    //返回线段所求点坐标
}}
```

- 第33-51行为计算圆弧的n等分点坐标的方法，此方法是从正二十面体顶点坐标拓展出几何球顶点坐标的关键。
- 第54-63行为计算线段的n等分点坐标的方法，此方法是从正二十面体顶点纹理坐标拓展出几何球顶点纹理坐标的关键。

> **提示**
>
> 在devideBall方法中用到的MyMathUtil为求解n元一次方程组的工具类，其采用的是doolittle（杜利特尔）分解算法。由于对此问题的详细讨论超出了本书所设定的范围，因此对此类不做详细介绍，需要的读者请自行查阅随书光盘中的源代码。

8.6　足球碳分子模型的搭建

前面五节介绍了几种基本形状的构建，本节将介绍用两种基本形状搭建足球碳（C_{60}）分子模型。由于足球碳分子的几何结构比较复杂，因此本节首先介绍搭建足球碳分子几何结构的基本原理，然后再给出一个实现的案例Sample8_6。

8.6.1　搭建的基本原理

学过中学化学的读者都知道，足球碳分子是由60个碳原子组成的，几何结构就像一个足球，也就是由很多正五边形和正六边形构成。本节案例的实现中，每个正五边形和正六边形的顶点处都会放置一个圆球，而每条边则会放置一根圆柱。

1．足球碳的整体开发思路

搭建立体的足球碳之前，首先需要了解一下足球的平面展开图，如图8-22所示，它由正五边形和正六边形组成，从中心的正五边形向外发散产生5个相同分支，只是在其中一个分支的末尾多添加了一个正五边形。

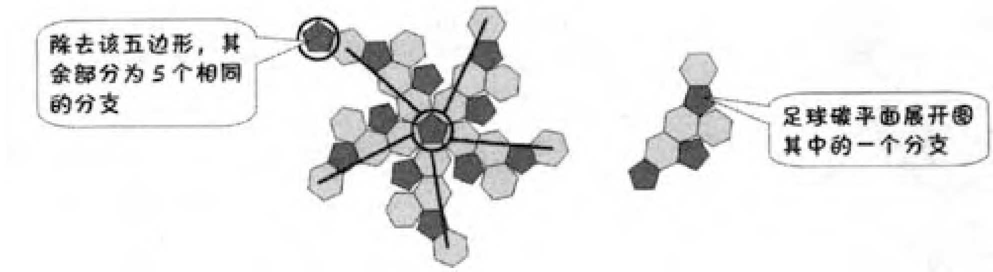


图8-22　足球碳的平面展开图

从图8-22中可以看出，足球碳分子模型的几何结构是由右侧的基本单元以不同的位置、姿态重复出现5次而构成的。当然要注意到的是，最后还需要补充一个正五边形。因此，搭建足球碳分子模型的工作可以化简为搭建图8-22右侧的基本单元。

2．旋转向量法构建正多边形

从前面的介绍中已经知道，搭建足球碳的关键是搭建其中的一个分支。构建一个分支的基本单元为正五边形及正六边形。下面以正五边形为例，介绍如何搭建这两种基本单元。读者可能会想到，直接给出正五边形各个顶点的坐标即可。但那样计算还是比较复杂，本节案例采用的是基于基本变换的方式，具体过程如图8-23所示。


图8-23　通过基本变换构建正五边形

从图8-23中可以看出，搭建的基本过程为。

- 首先给出由一个圆球与一根圆柱构成的正五边形的一部分（后面简称为一条边），如图8-23 中最左侧的小图所示。
- 然后将坐标系平移到圆柱右侧的下一个顶点的位置，并将坐标系旋转72°，接着再绘制一条边，形成如左起第二幅图的几何结构。
- 重复上面步骤3次，即可形成如最右边小图所示的正五边形。

提示

正六边形的构建思路与正五边形完全相同，只是每次旋转的角度是60°，这里不再赘述。

3．足球碳立体图形构建过程中一次重要的旋转

前面的平面展开图给出了构成基本单元的正五边形与正六边形之间的基本关系，但在实际的足球碳中，这些正五边形与正六边形并不是位于同一个平面上。因此用正五边形与正六边形构建基本单元时还需要考虑到这一点，下面会介绍具体的处理方法。

为了表述的方便，首先给基本单元中的第一个正五边形及与之相邻的正六边形的顶点编上号，具体情况如图8-24所示。

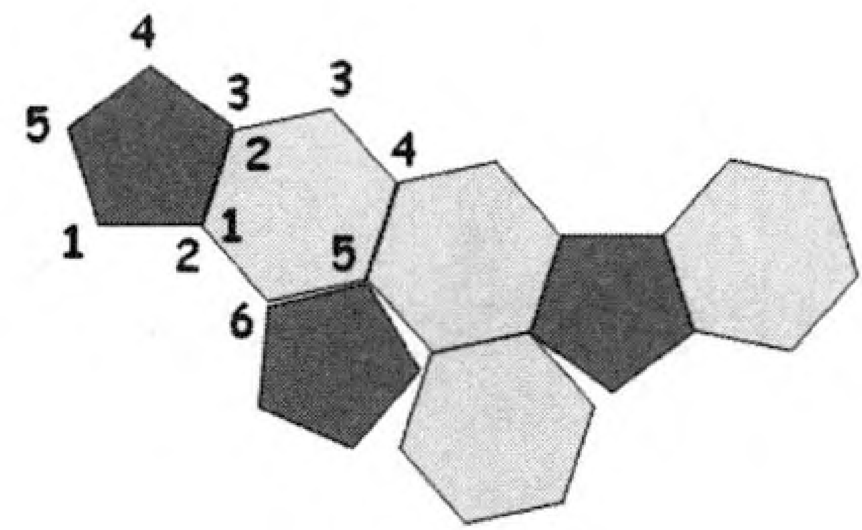


图8-24　正多边形的编号方式

从前面的介绍中可以想到，由于正五边形与正六边形不是共面的，因此当构建正六边形时完成了1-2边的构建后，需要将坐标系绕1-2边旋转一定的度数再去构建其他边，具体情况如图8-25所示。

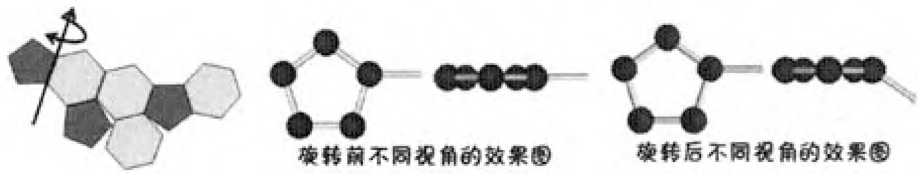


图8-25　旋转前后的视觉效果图

旋转的角度为39°，足球碳分子各平面间的二面角都是如此。构建完此正六边形后，继续重复上述做法即可得到一个基本单元的实际几何结构。构建出一个基本单元后，将该基本单元进行相应的旋转便可以得到完整的足球碳分子几何结构了，如图8-26所示。

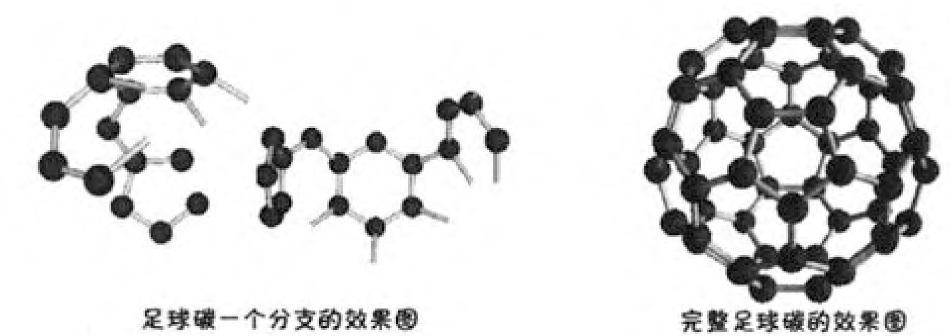


图8-26　足球碳的一个分支

说明

从图8-26的左图可以看出构造出的足球碳基本单元中部分五边形与六边形并不完整，这是因为实际上分支与分支之间有共用边，这些边仅绘制一次即可（这一点从图8-22中可以看出）。另外，出于同样的原因。平面展开图中最后需要补充的正五边形实际上也是不需要的，因为其每条边都是与某个基本单元的共用边。

8.6.2　案例的开发

上一小节介绍了搭建足球碳几何结构的基本原理，本小节将给出一个具体的实现案例Sample8_6，其运行效果如图8-27所示。

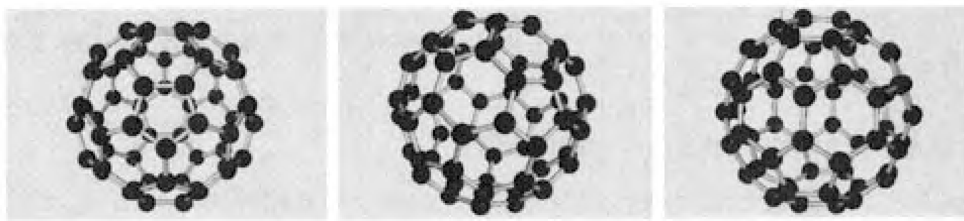


图8-27　案例Sample8_6运行效果图

提示

由于本书的插图采用灰度印刷，因此看起来效果不是很好。请读者用真机自行运行本案例观察，效果会好很多。

了解了本案例的运行效果后，就可以进行代码的开发了。由于本案例中用到的圆柱与圆球在前面的章节中已经介绍过，因此这里仅给出本案例中有代表性的用于搭建足球碳几何结构的相关代码，具体内容如下。

（1）首先给出的是用于搭建正多边形的RegularPolygon类，其代码如下。

代码位置：见随书光盘中源代码/第8章/Sample8_6/com/bn/Sample8_6目录下的RegularPolygon.java。

```
package com.bn.Sample8_6;                                  //包声明
import java.util.ArrayList;   import java.util.List;      //相关类的引入
public class RegularPolygon  {
   float length;                                           //边的长度
   ....//此处省略了部分声明成员变量的代码，读者可自行查阅随书光盘中的源代码
   public RegularPolygon(MySurfaceView mv,int borderCount,double angle,
     float length,double[] initPoint,double[] initVector,double[] pivot,int[] vertices,
     int[] borders){
    ....//此处省略了部分成员变量赋值的代码，读者可自行查阅随书光盘中的源代码
    this.pivot = pivot;//父对象的旋转轴，本对象的旋转轴在父对象的旋转轴的基础上旋转得到
        children = new ArrayList<RegularPolygon>(); //存放RegularPolygon类对象的列表
        this.verticesList = Utils.getRegularPentagonVertexData(//获取多边形顶点坐标列表
                initPoint, initVector, length,angle,vectors,borderCount,pivot);
      }
      public void drawSelf(float xOffset,float yOffset) {  //绘制正多边形的方法
       for(int i=0;i<vertices.length;i++){                 //依次绘制各个顶点位置的球
           int index = vertices[i];                        //顶点的索引
           float x = verticesList.get(3*index);            //获取该点的坐标值
           float y = verticesList.get(3*index+1);
           float z = verticesList.get(3*index+2);
           MatrixState.pushMatrix();                       //保护现场
           MatrixState.translate(x, y, z);                 //移动到顶点的位置
           mv.ball.drawSelf();                             //绘制球
```

```
            MatrixState.popMatrix();                        //恢复现场
        }
        for(int i=0;i<borders.length;i++){                  //依次绘制代表边的圆柱
            int index = borders[i];                         //获取索引
            float x = verticesList.get(3*index);            //获取边的起点坐标
            float y = verticesList.get(3*index+1);
            float z = verticesList.get(3*index+2);
            double[] vector = vectors[index];               //获取边的向量
            MatrixState.pushMatrix();                       //保护现场
            MatrixState.translate(x, y, z);                 //首先移动到起点
            MatrixState.pushMatrix();                       //保护现场
            Utils.moveXToSomeVector(vector);                //x 轴变换到指定向量的坐标系
            MatrixState.translate(Constant.LENGTH/2, 0, 0); //平移变换
            mv.stick.drawSelf();                            //绘制代表边的圆柱
            MatrixState.popMatrix();                        //恢复现场
            MatrixState.popMatrix();                        //恢复现场
        }
        drawChildren(xOffset, yOffset);            //绘制同属一个基本单元的其他多边形
    }
    public RegularPolygon buildChild(int borderCount,double angle,
            int position,int[] vertices,int[] borders){
                                        //创建与此多边形同属一个基本单元的相邻多边形
        double[] initPoint = new double[3];                 //创建顶点数组
        for(int i=0;i<3;i++){                               //为顶点数组赋值
            initPoint[i]=verticesList.get(3*position+i);
        }
        double[] initVector = vectors[position];            //获取方向向量
        double[] tempPivot = Utils.copyVecor(this.pivot);   //得到旋转轴
        RegularPolygon child = new RegularPolygon(this.mv,  //创建多边形
            borderCount, angle, length, initPoint, initVector,tempPivot,vertices,
            borders);
        children.add(child);                                //加入到 children 列表中
        return child;                                       //返回新建的多边形对象
    }
    private void drawChildren(float xOffset,float yOffset){
                                           //绘制同属一个基本单元的其他多边形
        for(int i=0;i<children.size();i++){        //依次绘制列表中的其他多边形
            RegularPolygon child = children.get(i);//从列表中取出一个多边形
            MatrixState.pushMatrix();              //保护现场
            child.drawSelf(xOffset, yOffset);      //绘制一个多边形
            MatrixState.popMatrix();}}             //恢复现场
}
```

- 上述RegularPolygon类对象是搭建足球碳几何结构的基石，每个RegularPolygon类对象代表一个正多边形。根据创建时所给参数的不同可以是正五边形，也可以是正六边形。
- 此类中的buildChild方法也很重要，其用于构建与此多边形相邻的其他多边形。这样只要有了基本单元中的第一个正五边形后就可以方便地生成其他正多边形了。

（2）接着给出的是MySurfaceView类中进行初始化工作的onSurfaceCreated方法，其代码如下。

代码位置：见随书光盘中源代码/第8章/Sample8_6/com/bn/Sample8_6目录下的MySurfaceView.java。

```
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    GLES20.glClearColor(0.9f,0.9f,0.9f, 1.0f);              //设置屏幕背景色 RGBA
    GLES20.glEnable(GLES20.GL_DEPTH_TEST);                    //启用深度测试
    GLES20.glEnable(GLES20.GL_CULL_FACE);                     //设置为打开背面剪裁
    MatrixState.setInitStack();                               //初始化变换矩阵
    float[] colorValue = {1,0,0,1};                           //创建球对应的颜色值数组
    ball = new Ball(MySurfaceView.this,Constant.BALL_R,colorValue);//创建球对象
    colorValue = new float[]{1,1,0,1};                        //创建圆柱对应颜色值数组
    stick = new Stick(MySurfaceView.this,Constant.LENGTH,
            Constant.R,Constant.ANGLE_SPAN,colorValue); //创建圆柱对象
        double[] initPoint = Utils.getFirstPoint(Constant.LENGTH);
                                                          //得到第一个五边形左下点的坐标
        double[] initVector={1,0,0,1};                        //初始化方向向量
        double[] zPivot = {0,0,1,1};                          //以 z 轴为旋转轴
        int[] vertices = {0,1,2,3,4};  //此正五边形中需要绘制的球的编号
        int[] borders = {0,1,2,3,4};   //此正五边形中需要绘制的圆柱的编号
        first = new RegularPolygon(MySurfaceView.this, 5,72 ,
            Constant.LENGTH, initPoint, initVector,zPivot,vertices,borders);
                                       //创建正五边形对象
        vertices = new int[]{2,3,4};    //与上一个正五边形相邻的正六边形中需要绘制的球的编
        borders = new int[]{1,2,3,4};  //号以及圆柱的编号，少了几个是因为重复的不需要绘制
        RegularPolygon rp2 = first.buildChild( 6, -60,1,vertices,borders);
                                       //创建正六边形对象
        vertices = new int[]{2,3,4,5}; //与上一个正六边形相邻的正六边形中需要绘制的球的编
        borders = new int[]{1,2,3,4,5};//号以及圆柱的编号，少了几个是因为重复的不需要绘制
        RegularPolygon rp4 = rp2.buildChild( 6, 60,3,vertices,borders);//创建正六边形对象
        vertices = new int[]{};        //与上一个正六边形相邻的正六边形中需要绘制的球的编
        borders = new int[]{1,5};      //号以及圆柱的编号，少了几个是因为重复的不需要绘制
        rp4.buildChild( 6, -60,2,vertices,borders);            //创建正六边形对象
        vertices = new int[]{2};       //与上一个正六边形相邻的正五边形中需要绘制的球的编
        borders = new int[]{1,2};      //号以及圆柱的编号，少了几个是因为重复的不需要绘制
        RegularPolygon rp6 = rp4.buildChild( 5, -72,3,vertices,borders);//创建正五边形对象
        vertices = new int[]{3,4,5};   //与上一个正五边形相邻的正六边形中需要绘制的球的编
        borders = new int[]{2,3,4,5};  //号以及圆柱的编号，少了几个是因为重复的不需要绘制
        rp6.buildChild( 6, 60,2,vertices,borders);        //创建正六边形对象
  }}
```

说明

上述代码主要是创建了构成图8-24所示的基本单元的各个多边形。细心的读者可能会发现不但创建某些多边形时舍弃了一些边或顶点，而且还有一些多边形被整体省略了，这都是因为绘制时不需要重复绘制位于同一位置的顶点和边。

（3）下面给出的是进行一些旋转及向量计算的工具类Utils，其代码如下。

代码位置：见随书光盘中源代码/第8章/Sample8_6/com/bn/Sample8_6目录下的Utils.java。

```
package com.bn.Sample8_6;                          //包的声明
```

```java
import java.util.ArrayList;    import java.util.List;//相关类的引入
public class Utils {
    static boolean first=true;                          //是否为第一个绘制的多边形的标志位
    ……//此处省略了向量规格化的 normalizeVector 方法
    ……//此处省略了求向量模的 module 方法
    public static double[] nRotate(double angle,double n[],double gVector[])
                                                        //绕任意轴旋转的方法
    {.....//该方法将会在后面的内容中详细介绍}
    public static List<Float> getRegularPentagonVertexData(double[] initPoint,double[]
    initVector,
            double length,double angle,double[][] vectors,int borderCount,double[] pivot)
        {.....//该方法将会在后面的内容中详细介绍}
        public static double[] copyVecor(double[] vector){  //复制向量数组
            double[] copy = new double[vector.length];      //创建复制数组
            for(int i=0;i<vector.length;i++){  //依次将源数组中的值复制进目标数组
                copy[i]=vector[i];
            }
            return copy;                                    //返回复制的数组
        }
        public static int compare(double x,double y){   //比较两个数是否约等于的方法
            if(Math.abs(x-y)<0.000001){         //如果两个数相减，绝对值小于阈值
                return 0;                       //返回 0，表示两个数基本相等
            }else if(x-y>0.000001){             //如果大于阈值
                return 1;                       //返回 1，表示第 1 个参数大于第 2 个参数
            }else{
                return -1;                      //否则返回-1，表示第 2 个参数大于第 1 个参数
        }}
        public static double[] getFirstPoint(float length){
                                                //计算第一个五边形左下角点坐标的方法
            double first[] = new double[3];     //X、Y、Z 坐标值数组
            first[0]=-length/2;                 //x 坐标值
            first[1]=-length/(2*Math.tan(Math.toRadians(36)));  //y 坐标值
            first[2]=0;                         //由于在 xy 平面上，z 值自然为 0
            return first;
        }
        public static double getAngleBetweenTwoVector(double[] vector1,double[] vector2){
            double angle=0;
            //计算两个向量的点积
            double DJ = vector1[0]*vector2[0]+vector1[1]*vector2[1]+vector1[2]*
            vector2[2];
            double mode = getMode(vector1)*getMode(vector2);//求两个向量模的积
            double cosa = DJ/mode;                          //计算两个向量夹角的余弦值
            if(compare(cosa,1)==0){                         //如果结果为 1
                return 0;                                   //返回角度为 0
            }
            angle = Math.toDegrees(Math.acos(cosa));        //计算角度值
            return angle;                                   //返回角度值
        }
        public static double getMode(double[] vector){      //求向量的模
            return Math.sqrt(vector[0]*vector[0]+vector[1]*vector[1]+vector[2]
```

```
            *vector[2]);
        }
        ……//此处省略了求向量叉积的getCJ方法，读者可自行查阅随书光盘的源代码
        public static void moveXToSomeVector(double[] vector){
                                                //将X轴转到与指定向量方向相同的方法
            double x[]={1,0,0};                              //X轴向量
            double angle = getAngleBetweenTwoVector(x,vector);//指定向量与x轴的夹角
            double pivot[] = getCJ(x,vector);//将X轴向量与指定向量叉积以得到旋转轴向量
           MatrixState.rotate((float)angle, (float)pivot[0], (float)pivot[1],(float)
            pivot[2]);
}}
```

说明

上述Utils类中主要是给出了一些辅助计算的工具方法，将这些工具方法单独放到工具类中有利于代码重用，并可以提高代码的可维护性。

（4）下面给出的是Utils类中的nRotate方法，具体内容如下。

代码位置：见随书光盘中源代码/第8章/Sample8_6/com/bn/Sample8_6目录下的Utils.java。

```java
public static double[] nRotate(double angle,double n[],double gVector[]){
    angle = Math.toRadians(angle);                              //转换为弧度值
    double[][] matrix={                                         //绕任意轴旋转变换的矩阵
      {n[0]*n[0]*(1-Math.cos(angle))+Math.cos(angle),n[0]*n[1]*(1-Math.cos(angle))+
         n[2]*Math.sin(angle),n[0]*n[2]*(1-Math.cos(angle))-n[1]*Math.sin(angle),0},
      {n[0]*n[1]*(1-Math.cos(angle))-n[2]*Math.sin(angle),n[1]*n[1]*(1-Math.cos(angle))+
             Math.cos(angle),n[1]*n[2]*(1-Math.cos(angle))+n[0]*Math.sin(angle),0},
      {n[0]*n[2]*(1-Math.cos(angle))+n[1]*Math.sin(angle),n[1]*n[2]*(1-Math.cos(angle))-
                 n[0]*Math.sin(angle),n[2]*n[2]*(1-Math.cos(angle))+Math.cos
                 (angle),0},
        {0,0,0,1}  };
      double[] tempDot={gVector[0],gVector[1],gVector[2],gVector[3]};//方向向量赋值
      for(int j=0;j<4;j++){                                     //旋转矩阵和向量相乘
          gVector[j]=(tempDot[0]*matrix[0][j]+tempDot[1]*matrix[1][j]+
                     tempDot[2]*matrix[2][j]+tempDot[3]*matrix[3][j]);}//向量相乘
      return gVector;                                           //返回结果
  }
```

说明

该方法的作用为在生成多边形下一条边的方向向量过程中，根据旋转的角度值、上一条边的方向向量计算出要绘制的边的方向向量。

（5）最后给出的是getRegularPentagonVertexData方法，其代码如下。

代码位置：见随书光盘中源代码/第8章/Sample8_6/com/bn/Sample8_6目录下的getRegularPentagonVertexData.java。

```
public static List<Float> getRegularPentagonVertexData(double[] initPoint,double[] initVector,
        double length,double angle,double[][] vectors,int borderCount,double[] pivot){
    List<Float> verticesList = new ArrayList<Float>();  //新建一个ArrayList
    double[] startPoint=initPoint;                       //起点坐标
```

```java
    double[] endPoint;                                          //终点坐标
    double[] vector = copyVecor(initVector);                    //复制第一条边的方向向量
    int index=0;
    double[] vectorS = copyVecor(vector);                       //将向量复制一份
    vectors[index++]=vectorS;                                   //将第一个向量保存
        for(int i=0;i<initPoint.length;i++){             //将第一个点的坐标依次添加到列表中
            verticesList.add((float) initPoint[i]);
        }
        while(index<borderCount){                                   //循环计算其余的点的坐标
            endPoint = new double[3];                               //创建当前线段的终点
            endPoint[0]=startPoint[0]+length*vector[0];      //计算终点 x 坐标
            endPoint[1]=startPoint[1]+length*vector[1];      //计算终点 y 坐标
            endPoint[2]=startPoint[2]+length*vector[2];      //计算终点 z 坐标
            if(               //如果计算出来的终点等于第一个点，则计算完毕，退出循环
                compare(endPoint[0],initPoint[0])==0        //调用 compare 方法进行比较
                && compare(endPoint[1],initPoint[1])==0
                && compare(endPoint[2],initPoint[2])==0){
                break;
            }
            for(int i=0;i<endPoint.length;i++){                 //将终点的坐标添加到列表中
                float value = (float) endPoint[i];
                if(Math.abs(value)<0.000001){                   //如果结果为一个很小的数
                    verticesList.add(new Float(0.0f));      //那么向列表中放入 0
                    continue;                                   //继续下一次循环
                }else{
                    verticesList.add((float) endPoint[i]); //向列表中加入数据
            }}
            if(index==1){                                   //计算下一条边的方向向量
                vector = nRotate(angle,pivot,vector);   //绕旋转轴生成第二个向量
                if(!first){                                 //如果不是第一个多边形
               double tempAngle = 39*angle/Math.abs(angle);
                vector = nRotate(tempAngle,initVector,vector);//将第二个向量绕第一个向量旋转
                pivot = nRotate(tempAngle,initVector,pivot);//生成新的旋转轴
            }
                first=false;
            }else{
                vector = nRotate(angle,pivot,vector);//将当前的方向向量旋转得到新的方向向量
            }
            vectorS = copyVecor(vector);                    //将向量复制一份
            vectors[index++]=vectorS;                       //将新的向量保存
            startPoint = endPoint;                          //将当前线段的终点作为下条线段的起点
        }
        return verticesList;                                //返回列表结果
    }
```

> **说明**
>
> 该方法的作用为生成要绘制的一个多边形的顶点坐标数据，并生成当前多边形每条边的方向向量。为了便于理解该方法，请读者参考前面的图8-23和图8-25。

8.7 贝塞尔曲线及旋转面

前面几节介绍的都是某种具体几何形状的搭建，本节将向读者介绍旋转面的搭建，通过旋转面技术可以搭建出各种各样不同形状的物体。掌握了旋转面技术后，读者的造型能力将得到很大的提升。

8.7.1 三维旋转曲面的生成

旋转面搭建的基本原理非常简单，将特定曲线绕指定的轴旋转一周即可产生，如图8-28所示。

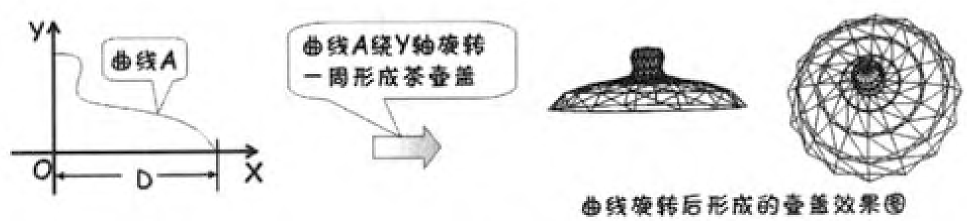


图8-28 旋转曲面茶壶盖的形成

从图8-28中可以看出，左侧的曲线绕Y轴旋转一周后产生了一个茶壶盖的形状。从上述场景中可以想象出，平时常见的茶壶、饮料瓶等轴对称的形状都可以采用旋转面技术轻松地生成。

这时读者可能会有这样的问题：曲线如何产生呢？总不至于像画抛物线那样描点做图，再将数据从坐标纸上读出录入源代码吧？笔者当然不会让读者去采用那么原始的做法，因此下一小节将向读者介绍一种很常用的曲线生成算法。

8.7.2 贝塞尔曲线

通过上一小节的介绍，读者应该已经了解到了采用旋转面生成轴对称几何形状的方便之处。但从旋转面技术的特点也可以看出，要想形成所需的旋转面，必须先得到对应的曲线。因此，本节将向读者介绍一种非常常用的曲线生成技术——贝塞尔曲线。

贝塞尔曲线首先由Paul de Casteljau于1959年发明，后来又由Pierre Bezier在1962年左右独立发明。生成贝塞尔曲线的算法有很多种，本节介绍的是简洁精致的德卡斯特里奥算法。下面将从最简单的3个点开始，逐渐推导出n个点的贝塞尔曲线解析方程。

1．三个点的贝塞尔曲线

首先需要给出3个控制点，分别为P_0、P_1、P_2。然后选定一个值t（t在0到1之间），通过t在P_0P_1线段上确定一个点A，使得线段P_0A的长度与线段AP_1的长度的比值为t:1-t。以同样的方式在P_1P_2线段上也确定一个点B，那么点A和点B的坐标可以用下面的参数方程来表示。

$A(t)=(1-t)P_0+tP_1$　　(1)

$B(t)=(1-t)P_1+tP_2$　　(2)

确定了A点与B点的位置后，在线段AB上也按照此方式可以确定点P，其坐标可以用下面的参数方程表示。

$P(t)=(1-t)A+tB$　　(3)

将式（1）、式（2）带入式（3）中，可以得到下面的参数方程。

$P(t)=(1-t)^2P_0+2t(1-t)P_1+t^2P_2$　　(4)

对t连续地取0～1的数，可以得到P点的轨迹。这个轨迹就是由3个点确定的贝塞尔曲线，如图8-29所示。

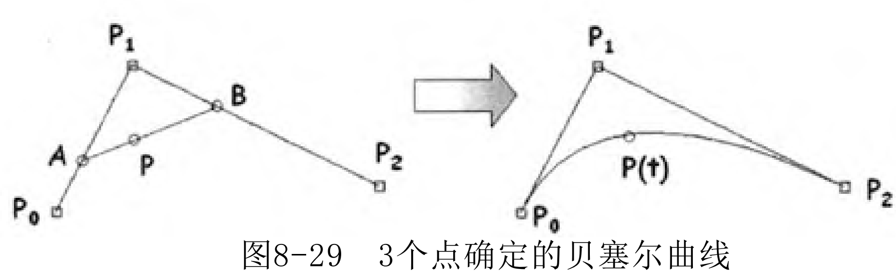


图8-29　3个点确定的贝塞尔曲线

进一步分析参数方程可以发现，3个点确定的贝塞尔曲线实际上是一条抛物线。当点数增加后就大大不同了，下面的图8-30给出了几种

不同情况下4个点确定的贝塞尔曲线。

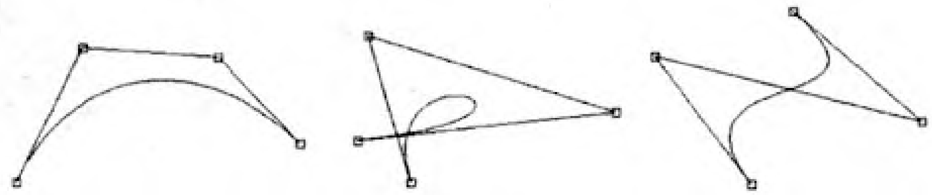


图8-30　基于4个点的贝塞尔曲线

根据前面介绍的基于3个点生成贝塞尔曲线的方法，很容易地可以推导出4个点确定的贝塞尔曲线参数方程如下。

$$P(t)=(1-t)^3P_0+3t(1-t)^2P_1+3t^2(1-t)^2P_2+t^3P_3$$

2．任意个点的贝塞尔曲线

进一步把前面介绍的4个点确定贝塞尔曲线的参数方程进行扩展，可以得到有L+1个控制点的贝塞尔曲线的参数方程，具体内容如下。

$$P(t)=\sum_{k=0}^{L}\frac{L!}{k!(L-k)!}(1-t)^{L-k}t^kP_k$$

从上述参数方程中可以看出，只要对t连续地取0～1的数，得到的P点轨迹即为这L+1个点确定的贝塞尔曲线。通过修改控制点的数量、位置可以得到任意所需的曲线，如图8-31所示。

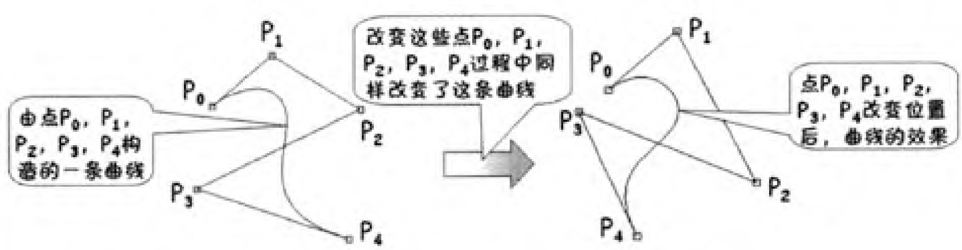


图8-31　基于5个点的贝塞尔曲线

说明

图8-31中展示了采用德卡斯特里奥算法基于5个点生成贝塞尔曲线的效果图，从左右两幅图的对比中可以看出，通过改变控制点的位置可以得到不同形状的曲线。

8.7.3 Bezier曲线生成工具

上一小节介绍了基于任意个点生成贝塞尔曲线的参数方程，理论上有了参数方程就可以方便地计算出曲线上各个点的位置了。但若是纯手工操作，几乎是一个不可能完成的任务。因此本小节将介绍一个基于上一小节介绍的参数方程生成贝塞尔曲线的小工具。此小工具为笔者自己开发，需要的读者可以直接使用，其运行效果如图8-32所示。

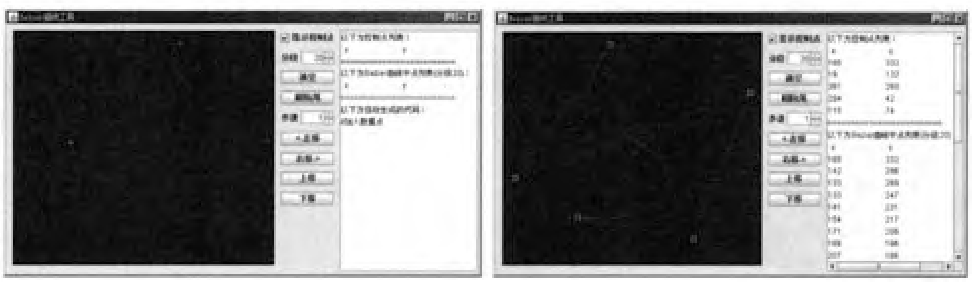


图8-32 运行效果图

说明

从图8-32中可以看出，这个小工具采用的是所见即所得的方法来设计贝塞尔曲线。只需要用鼠标光标在空白处单击即可增加新的控制点，用鼠标光标还可以拖拉控制点以改变曲线的形状，读者可以自行运行本软件体会。

了解了本软件的运行效果后，下面将简要介绍一下本软件的开发。由于Java Swing界面的开发超出了本书的讨论范围，因此这里仅介绍与贝塞尔曲线生成算法密切相关的代码，而略去Swing界面开发部分的代码，需要的读者请自行参考随书光盘中的源代码。

提示

对于只是想使用本软件生成贝塞尔曲线的数据为后面生成旋转面服务，并不想深入了解本软件如何开发的读者，可以略过此部分。

（1）首先给出的是代表贝塞尔曲线上每个点的类——BNPosition,其代码如下。

代码位置：见随书光盘中源代码/第8章/BezierEx/com/bn/bezier目录下的BezierEx.java。

```
package com.bn.bezier;                          //包声明
public class BNPosition {
    int x;                                      //曲线上点的x坐标
    int y;                                      //曲线上点的y坐标
    public BNPosition(float x,float y){
        this.x=(int) x;                         //x坐标赋值
        this.y=(int) y;                         //y坐标赋值
}}
```

提示

该类的一个对象代表生成的贝塞尔曲线上的一个点，生成贝塞尔曲线的主要工作就是生成代表曲线上不同点的此类对象的序列。

（2）接着给出的是根据控制点生成贝塞尔曲线上点序列的BezierUtil类，其代码如下。

代码位置：见随书光盘中源代码/第8章/BezierEx/com/bn/bezier目录下的BezierUtil.java。

```
package com.bn.bezier;                                            //包声明
import java.util.ArrayList;                                       //相关类的引入
public class BezierUtil {
  static ArrayList<BNPosition> al=new ArrayList<BNPosition>();    //控制点的列表
  public static ArrayList<BNPosition> getBezierData(float span){
                                                                  //生成贝塞尔曲线上点序列的方法
      ArrayList<BNPosition> result=new ArrayList<BNPosition>(); //存放结果点序列的列表
      int n=al.size()-1;                                          //得到控制点线段数
      if(n<1){                                                    //线段数少于1，无贝塞尔曲线
          return result;                                          //返回空列表
          }
          int steps=(int) (1.0f/span);                            //计算总分段数
          long[] jiechengNA=new long[n+1];                        //声明一个长度为n+1的阶乘数组
          for(int i=0;i<=n;i++){                                  //求0到n的阶乘
              jiechengNA[i]=jiecheng(i);                          //调用jiecheng方法计算i的阶乘
          }
          for(int i=0;i<=steps;i++){                              //分段进行循环
              float t=i*span;
              if(t>1){                                            //t的值必须在0~1
                  t=1;
              }
              float xf=0;                                         //贝塞尔曲线上点的x坐标
              float yf=0;                                         //贝塞尔曲线上点的y坐标
              float[] tka=new float[n+1];                         //新建一个长度为n+1的数组
              float[] otka=new float[n+1];                        //新建一个长度为n+1的数组
              for(int j=0;j<=n;j++){
                  tka[j]=(float) Math.pow(t, j);                  //计算t的j次幂
                  otka[j]=(float) Math.pow(1-t, j);//计算1-t的j次幂
              }
              for(int k=0;k<=n;k++){          //循环n+1次计算贝塞尔曲线上各个点的坐标
                  float xs=(jiechengNA[n]/(jiechengNA[k]*jiechengNA[n-k]))*tka[k]*
                  otka[n-k];
                  xf=xf+al.get(k).x*xs;
                  yf=yf+al.get(k).y*xs;
              }
              result.add(new BNPosition(xf,yf));                  //将得到的点存入结果列表
          }
          return result;                                          //返回贝塞尔曲线上点的列表
      }
      public  static long jiecheng(int n){                        //求阶乘的方法
          long result=1;                                          //声明一个long型的变量
```

```
        if(n==0){                              //0 的阶乘为 1
            return 1;
        }
        for(int i=2;i<=n;i++){                 //求大于等于 2 的数的阶乘
            result=result*i;
        }
        return result;                         //返回阶乘的结果值
}}
```

说明

上述代码主要是基于前面介绍的参数方程依次对t取0-1的值得到了所生成的贝塞尔曲线上系列点的坐标。请读者特别注意的是，对多次计算参数方程一直要用到的一些相同项预先进行了计算，并将结果存入了数组。这样的好处是避免了不必要的重复计算，提高速度，读者以后也可以在开发中使用这种技巧。

8.7.4　印度古典建筑场景的开发

从前面的图8-32中可以看出，用笔者开发的工具软件生成贝塞尔曲线时不但有界面左侧所见即所得的部分，右侧还给出了贝塞尔曲线控制点的坐标列表。此坐标列表就是用于生成旋转面的数据，下面将给出一个基于软件生成的数据生成旋转面的案例Sample8_7，其运行效果如图8-33所示。

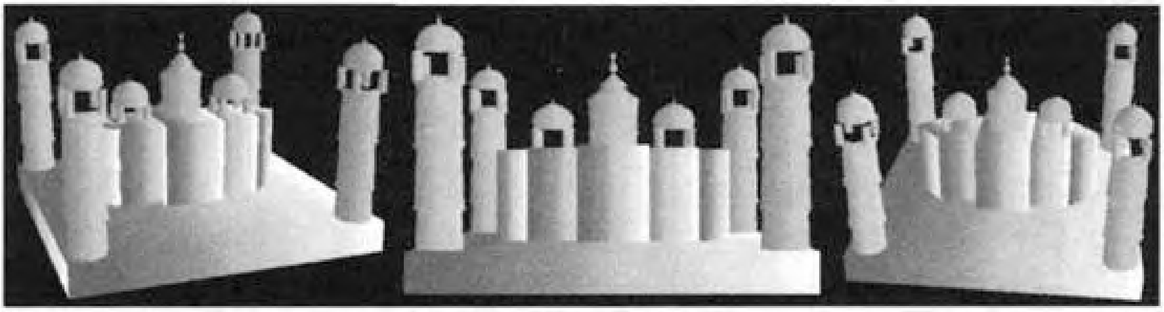


图8-33　运行效果图

> **说明**
>
> 从图8-33中可以看出，本案例基于贝塞尔曲线生成旋转面，最终搭建了一个类似与泰姬陵的印度古典建筑场景。该案例建筑群中的每一个组件均由相应的贝塞尔曲线旋转生成。

了解了本案例的运行效果后，就可以进行代码的开发了。虽然场景建筑群中的组件个数很多，但开发的思路大同小异。因此这里仅给出比较有代表性的建筑物顶部组件顶点数据生成的相关代码，具体内容如下。

代码位置：见随书光盘中源代码/第8

章/Sample8_7/com/bn/Sample8_7目录下的TopPart1.java。

```
public void initVertexData(float scale, int nCol ,int nRow ){//初始化顶点数据的方法
    float angdegSpan=360.0f/nCol;
    vCount=3*nCol*nRow*2;                                    //顶点个数
    ArrayList<Float> alVertix=new ArrayList<Float>();        //原顶点列表（未卷绕）
    ArrayList<Integer> alFaceIndex=new ArrayList<Integer>();//组织成面的顶点的编号值列表
    BezierUtil.al.clear();                                   //清空控制点列表
    BezierUtil.al.add(new BNPosition(-1, 171));              //控制点坐标数据1
```

```
BezierUtil.al.add(new BNPosition(14, 191));                    //控制点坐标数据 2
BezierUtil.al.add(new BNPosition(17, 183));                    //控制点坐标数据 3
    BezierUtil.al.add(new BNPosition(5, 154));                 //控制点坐标数据 4
    BezierUtil.al.add(new BNPosition(31, 274));                //控制点坐标数据 5
    BezierUtil.al.add(new BNPosition(32, 243));                //控制点坐标数据 6
    BezierUtil.al.add(new BNPosition(30, 230));                //控制点坐标数据 7
    BezierUtil.al.add(new BNPosition(0, 253));                 //控制点坐标数据 8
    //根据控制点生成贝赛尔曲线上点的列表
    ArrayList<BNPosition> alCurve=BezierUtil.getBezierData(1.0f/nRow);
    for(int i=0;i<nRow+1;i++){//根据得到的曲线上点的列表生成旋转面上点的坐标列表
        double r=alCurve.get(i).x*Constant.DATA_RATIO*scale; //当前圆的半径
        float y=alCurve.get(i).y*Constant.DATA_RATIO*scale;  //当前 y 值
        for(float angdeg=0;Math.ceil(angdeg)<360+angdegSpan;angdeg+=angdegSpan){
            double angrad=Math.toRadians(angdeg);              //当前弧度
            float x=(float) (-r*Math.sin(angrad));             //计算顶点的坐标值
            float z=(float) (-r*Math.cos(angrad));             //计算顶点的坐标值
            alVertix.add(x); alVertix.add(y); alVertix.add(z);
                                                   //将顶点数据加入到列表中
    }}
    for(int i=0;i<nRow;i++){                           //通过循环计算出卷绕成三角形的
        for(int j=0;j<nCol;j++){                       //顶点编号列表
            int index=i*(nCol+1)+j;                    //当前四边形第一个顶点的编号
            alFaceIndex.add(index+1);                  //第一三角形 1 号点编号
            alFaceIndex.add(index+nCol+2);             //第一三角形 2 号点编号
            alFaceIndex.add(index+nCol+1);             //第一三角形 3 号点编号
            alFaceIndex.add(index+1);                  //第二三角形 1 号点编号
            alFaceIndex.add(index+nCol+1);             //第二三角形 2 号点编号
            alFaceIndex.add(index);                    //第二三角形 3 号点编号
    }}
    float[] vertices=new float[vCount*3];              //卷绕成三角形的顶点坐标数组
    vertices=VectorUtil.calVertices(alVertix, alFaceIndex);
    ......//此处省略生成顶点坐标数据缓冲的代码，读者可自行查看随书光盘中的源代码
    ArrayList<Float> alST=new ArrayList<Float>();      //顶点的纹理坐标列表
    float yMin=999999999;                                  //y 最小值
    float yMax=0;                                          //y 最大值
    for(BNPosition pos:alCurve){
        yMin=Math.min(yMin, pos.y);                        //y 最小值
        yMax=Math.max(yMax, pos.y);                        //y 最大值
    }
    for(int i=0;i<nRow+1;i++){
        float y=alCurve.get(i).y;                          //当前 y 值
        float t=1-(y-yMin)/(yMax-yMin);                    //t 坐标
        for(float angdeg=0;Math.ceil(angdeg)<360+angdegSpan;angdeg+=angdegSpan){
            float s=angdeg/360;                            //s 坐标
            alST.add(s); alST.add(t);                  //将计算出来的 ST 纹理坐标加入列表
    }}
    float[] textures=VectorUtil.calTextures(alST, alFaceIndex);
    ......//此处省略顶点数据缓冲的代码，读者可自行查看随书光盘的源代码
}
```

提示

上述代码中首先根据从工具软件获得的控制点坐标，同样也调用BezierUtil类中的工具方法生成贝塞尔曲线上点的坐标序列，然后根据生成的坐标序列生成旋转面上各个顶点的坐标以及纹理坐标，最后卷绕成三角形。

其他的建筑部件除了控制点不同外，代码的套路完全相同，这里就不一一赘述了，需要的读者请参考随书光盘中的源代码。

8.8 本章小结

本章向读者介绍了如何通过自己编程构建各种基本形状以及贝塞尔曲线旋转面，掌握了这些知识以后，读者搭建场景的能力应该有很大的提高。但现实世界中还有很多更为复杂的物体，如飞机、老鹰、茶壶等，若直接用代码搭建将会非常复杂，因此下一章会向读者介绍如何加载设计好的3D模型。

第9章 3D模型加载

前面章节所有案例中的3D模型都是采用直接给出顶点坐标值（如立方体）或基于数学公式用程序生成坐标值（如球体）的方式，这在一些简单的游戏场景中已经足够了。但如果要建立一些更加复杂、逼真的游戏场景，需要采用的物体其几何形状可能会很复杂而且不能直接用数学公式描述，如赛车游戏中的车、空战游戏中的战斗机等。

这种情况下一般首先用3D建模工具（如3DMax、Maya等）建立物体模型，然后导出成特定格式的模型文件并在应用程序中加载渲染。常用的3D模型文件格式有obj、3ds、fbx等，本章主要介绍obj模型文件的加载。

9.1 obj模型文件概览

介绍如何用程序加载obj模型文件之前，首先需要简单了解一下此类文件的格式及导出方式，本节将对这方面的内容进行简要的介绍。

9.1.1 obj文件的格式

obj文件是最简单的一种3D模型文件，其本质上就是文本文件，只是具有固定的格式而已。obj文件中将顶点坐标、三角形面、纹理坐标等信息以固定格式的文本字符串表示，下面给出了一个obj文件的片段。

```
# Max2Obj Version 4.0 Mar 10th, 2001
# Author wyf
#
v  -19.990179 -34.931675 -18.201921
v  20.111662 -34.931675 -18.201921
……
v  20.111662 27.748880 21.994425
# 8 vertices
vt  0.000000 0.000000 0.000000
vt  1.000000 0.000000 0.000000
……
vt  1.000000 1.000000 0.000000
# 12 texture vertices
vn  0.000000 0.000000 -1.570796
vn  0.000000 0.000000 -1.570796
……
vn  0.000000 0.000000 1.570796
# 8 vertex normals
g (null)
f 1/10/1 3/12/3 4/11/4
f 4/11/4 2/9/2 1/10/1
……
f 5/4/5 7/3/7 3/1/3
# 12 faces
g
```

从上述obj文件片段中可以看出，其内容是以行为基本单位进行组织的，每种不同前缀开头的 行有不同的含义，具体情况如下所列。

- “#”号开头的行为注释，在程序加载的过程中可以略过。
- “v”开头的行用于存放顶点坐标，其后面的3个数值分别表示一个顶点的X、Y、Z 坐标。

- “vt”开头的行用于存放顶点纹理坐标，其后面的3个数值分别表示纹理坐标的S、T、W分量。

> **提示**
>
> S、T纹理坐标读者已经非常熟悉了，W指的是深度纹理坐标，主要用于3D纹理的采样。由于OpenGL ES2.0中对3D纹理还没有普遍支持，需要特定的扩展，因此本书中的案例仅使用了S、T纹理坐标。

- “vn”开头的行用于存放顶点法向量，其后面的3个数值分别表示一个顶点的法向量在X轴、*Y* 轴、*Z* 轴上的分量。
- “g”开头的行表示一组的开始，后面的字符串为此组的名称。所谓组是指由顶点组成的一些面的集合。只包含“g”的行表示一组的结束，与“g”开头的行对应。
- “f”开头的行表示组中的一个面，如果是三角形（由于OpenGL ES仅支持三角形，故本书案例中采用的都是三角形）则后面有3组用空格分隔的数据，代表三角形的3个顶点。每组数据中包含3个数值，用“/”分隔，依次表示顶点坐标数据索引、顶点纹理坐标数据索引、顶点法向量数据索引。

> **说明**
>
> 例如有这样的一行“f200/285/200 196/280/196 195/279/195”，其表示三角形中3个顶点的坐标来自200、196、195号“v”开头的行，3个顶点的纹理坐标来自285、280、279号“vt”开头的行，3个顶点的法向量来自200、196、195号“vn”开头的行。计算行号时各种不同前缀是独立计算的，例如前面代码的第5行为“V”开头的2号行。

还有一点读者需要了解的是，obj文件中一般顶点坐标与面的数据是必须选择的，而法向量与纹理数据是可选的。

9.1.2　用3DMax设计3D模型

上一小节中介绍了obj文件的基本格式，可以看出虽然obj本质上是文本文件，但也不适合直接由人工录入其各方面的内容。很多的3D模型设计工具都可以导出obj格式的模型文件，笔者采用的是3DMax。3DMax是一款主流的、非常方便的3D建模软件，有着很高的市场占有率。用3DMax设计完3D模型后，可以非常方便地导出各种格式的模型文件，当然也包括obj文件。

下面简要介绍一下如何用3DMax导出obj模型文件，步骤如下所列。

（1）首先启动3DMax软件，并用其设计一个3D模型，如图9-1所示。

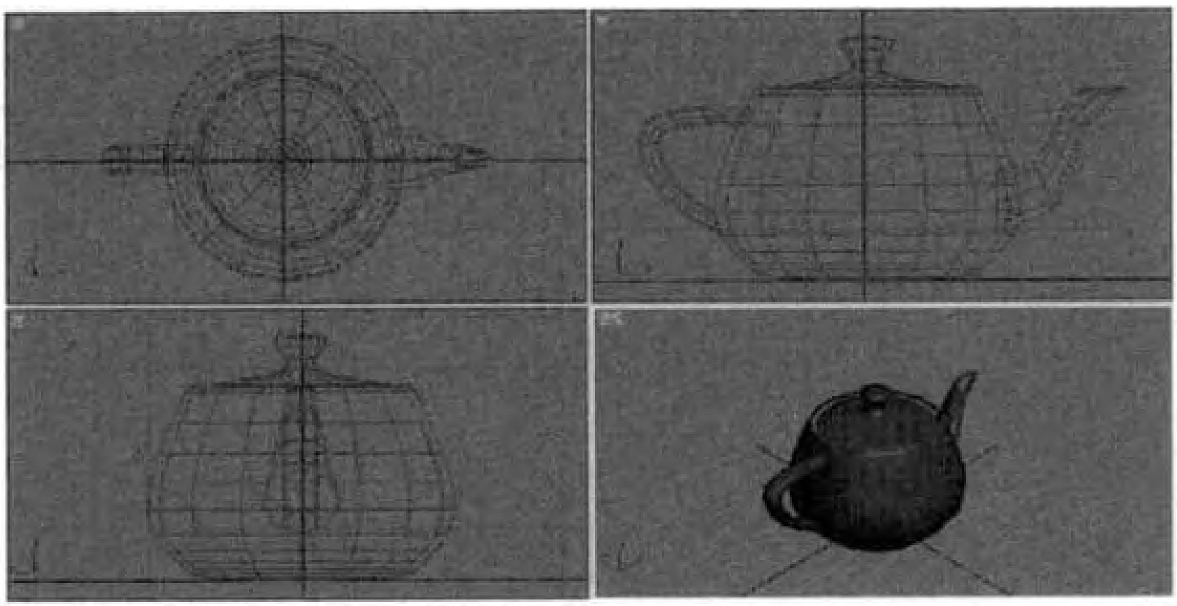


图9-1　用3DMax设计3D模型

（2）接着单击主菜单中的“文件”选项，弹出下拉菜单后，单击“导出”选项。如图9-2所示。


文件(F)
编辑(E)
工具(T)
组(G)
新建(N)...
Ctrl+N
重置(R)
打开(O)...
Ctrl+O
打开最近(T)
保存(S)
Ctrl+S
另存为(A)...
保存副本为(C)...
保存选定对象(D)...
外部参照对象(B)...
外部参照场景(C)...
文件链接管理器...
合并(M)...
合并动画...
替换(L)...
加载动画...
保存动画...
导入(I)...
导出(E)...
导出选定对象...

图9-2　选择“导出”选项

（3）单击“导出”选项后，系统将弹出“选择要导出的文件”对话框，输入期望的文件名，并将保存类型设置为“*.obj”，然后单击“保存”按钮，如图9-3所示。



图9-3　选择要导出的文件

（4）单击“保存”按钮后系统将弹出“对象导出器”对话框，首先选择面类型为“Triangles”，然后选中纹理坐标选项，最后单击“确定”按钮，即可完成obj文件的导出，如图9-4所示。

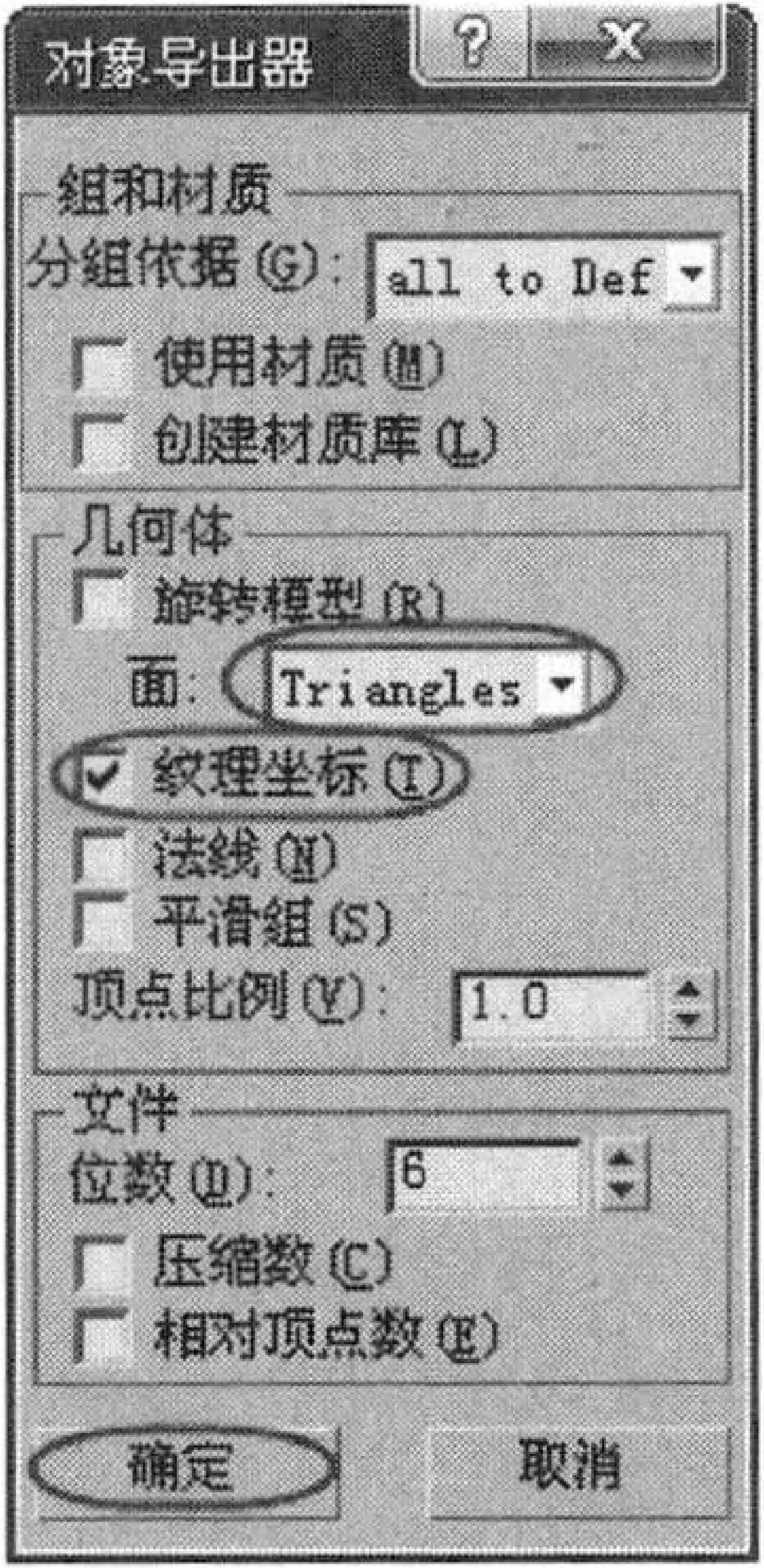

对象导出器
组和材质
分组依据(G):
all to Def
使用材质(M)
创建材质库(L)
几何体
旋转模型(R)
面:
Triangles
纹理坐标(T)
法线(N)
平滑组(S)
顶点比例(V):
1.0
文件
位数(D):
6
压缩数(C)
相对顶点数(E)
确定
取消

图9-4　对象导出器

提示

面类型一定要设置为“Triangles”，否则就不适合OpenGL ES平台的应用程序使用了。纹理坐标选项不一定要选中，需要根据渲染模型时是否使用纹理映射来决定。

9.2 加载obj文件

上一节介绍了如何从3DMax导出obj文件，本节将通过几个具体的案例来介绍如何将obj文件中的数据加载进应用程序中，并用OpenGL ES 2.0渲染呈现。

9.2.1 加载仅有顶点坐标与面数据的obj文件

本小节将给出一个非常简单的加载obj文件的案例Sample9_1，此案例中加载的obj文件仅有 顶点坐标与面数据，其运行效果如图9-5所示。

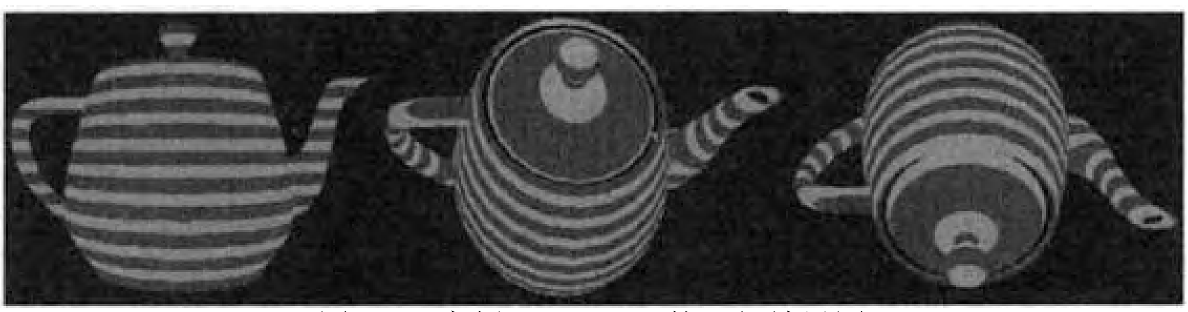


图9-5　案例Sample9_1的运行效果图

说明

从图9-5中可以看出，本案例渲染的是一个茶壶。可以想象，直接用程序自动生成其顶点坐标是非常困难的，因此模型的加载就成为3D应用程序开发人员必须掌握的技能。

了解了本案例的运行效果后，下面就可以进行代码的开发了。由于本案例与前面章节中很多的案例大部分代码是相同的，因此这里仅介绍本案例中新增的且具有代表性的部分，具体如下所列。

（1）首先介绍加载后用于渲染物体的LoadedObjectVertexOnly类，其代码如下。

代码位置：见随书光盘中源代码/第9章/Sample9_1/com/bn/Sample9_1目录下的LoadedObjectVertexOnly.java。

```java
package com.bn.Sample9_1;                                  //声明包
……//此处省略了部分类的引入代码，读者可自行查看随书光盘中的源代码
public class LoadedObjectVertexOnly {                      //加载后的物体——仅携带顶点信息
    int mProgram;                                          //自定义渲染管线着色器程序 id
    ……//此处省略了部分成员变量的声明，读者可自行查看随书光盘中的源代码
    FloatBuffer mVertexBuffer;                             //顶点坐标数据缓冲
    int vCount = 0;                                        //顶点数量
    public LoadedObjectVertexOnly(MySurfaceView mv, float[] vertices) {
        initVertexData(vertices);                          //初始化顶点坐标
            initShader(mv);                                //初始化着色器
        }
        public void initVertexData(float[] vertices) {     //初始化顶点坐标的方法
            vCount = vertices.length / 3;
            ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);
                                                           //创建顶点坐标数据缓冲
            vbb.order(ByteOrder.nativeOrder());            //设置字节顺序
            mVertexBuffer = vbb.asFloatBuffer();           //转换为 Float 型缓冲
            mVertexBuffer.put(vertices);                   //向缓冲区中放入顶点坐标数据
            mVertexBuffer.position(0);                     //设置缓冲区起始位置
        }
        ……//此处省略了 initShader 方法和 drawSelf 方法，读者可自行查看随书光盘的源代码
}
```

提示

该类非常简单，与前面章节中物体类不同的是，其顶点的位置数据是由构造器参数中传入的，而不是在该类中初始化顶点坐标的initVertexData方法中直接给出或计算出来的。

（2）开发完加载后用于渲染物体的LoadedObjectVertexOnly类后，接下来介绍从obj文件读入数据并创建该类对象的工具类——LoadUtil，其代码如下。

代码位置：见随书光盘中源代码/第9章/Sample9_1/com/bn/Sample9_1目录下的LoadUtil.java。

```
package com.bn.Sample9_1;                                          //声明包
……//此处省略了部分类的引入代码，读者可自行查看随书光盘中的源代码
public class LoadUtil {
    public static LoadedObjectVertexOnly loadFromFile (String fname, Resources r,
    MySurfaceView mv) {//从 obj 文件中加载顶点数据并生成 LoadedObjectVertexOnly 类对象
        LoadedObjectVertexOnly lo = null;                          //加载后物体的引用
        ArrayList<Float> alv = new ArrayList<Float>();             //原始顶点坐标列表
        ArrayList<Float> alvResult = new ArrayList<Float>();       //结果顶点坐标列表
        try {
                InputStream in = r.getAssets().open(fname);        //获取字节输入流
                InputStreamReader isr = new InputStreamReader(in);//将字节流转换成字符流
                BufferedReader br = new BufferedReader(isr);       //将字符流进一步封装
                String temps = null;
                while ((temps = br.readLine()) != null) {          //读取一行文本
                    String[] tempsa = temps.split("[ ]+");         //将文本行用空格符切分
                    if (tempsa[0].trim().equals("v")) {            //顶点坐标行
                        alv.add(Float.parseFloat(tempsa[1]));//将顶点的 x、y、z 坐标存入
                        alv.add(Float.parseFloat(tempsa[2]));  //原始顶点坐标列表
                        alv.add(Float.parseFloat(tempsa[3]));
                    } else if (tempsa[0].trim().equals("f")) {  //面数据行
                        int index = Integer.parseInt(tempsa[1].split("/")[0]) - 1;
                                                               //得到顶点编号
                        alvResult.add(alv.get(3 * index)); //将三角形第 1 个顶点的
                        alvResult.add(alv.get(3 * index + 1)); //x、y、z 坐标存入结果顶点
                        alvResult.add(alv.get(3 * index + 2)); //坐标列表
                        index = Integer.parseInt(tempsa[2].split("/")[0]) - 1;
                                                               //得到顶点编号
                        alvResult.add(alv.get(3 * index)); //将三角形第 2 个顶点的
                        alvResult.add(alv.get(3 * index + 1)); //x、y、z 坐标存入结果顶点
                        alvResult.add(alv.get(3 * index + 2)); //坐标列表
                        index = Integer.parseInt(tempsa[3].split("/")[0]) - 1;
                                                                   //得到顶点编号
                        alvResult.add(alv.get(3 * index));         //将三角形第 3 个顶点的
                        alvResult.add(alv.get(3 * index + 1));//x、y、z 坐标存入结果顶点
                        alvResult.add(alv.get(3 * index + 2)); //坐标列表
                }}
                int size = alvResult.size();                   //获取顶点坐标数量
                float[] vXYZ = new float[size];                //创建用于存储顶点坐标的数组
                for (int i = 0; i < size; i++) {
                    vXYZ[i] = alvResult.get(i);                //将顶点坐标数据转存到数组中
                }
                lo = new LoadedObjectVertexOnly(mv, vXYZ);  //创建物体对象
            } catch (Exception e) { Log.d("load error", "load error"); e.printStackTrace();}
            return lo;                                         //返回创建物体对象的引用
    }}
```

- 第4-5行中，参数fname为要加载的obj文件的名称，r为Resources对象的引用，mv为MySurfaceView对象的引用。
- 第10-33行通过输入流将顶点数据、面数据从obj文件中读入，并依照文件中的信息将顶点数据依次组织到结果顶点坐标列表中。
- 第34-39行首先将结果顶点坐标列表中的数据转存到数组中，并调用LoadedObjectVertexOnly类的构造器创建用于绘制的物体对象，最后将该对象的引用返回。

（3）介绍完LoadUtil工具类后，下面介绍用于渲染整个3D场景的MySurfaceView类，其代码如下。

代码位置：见随书光盘中源代码/第9章/Sample9_1/com/bn/Sample9_1目录下的MySurfaceView.java。

```java
package com.bn.Sample9_1;                                        //声明包
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
class MySurfaceView extends GLSurfaceView {
    private final float TOUCH_SCALE_FACTOR = 180.0f / 320;   //角度换算比例
    private SceneRenderer mRenderer;                          //场景渲染器
    private float mPreviousY;                                 //上次的触控位置的 Y 坐标
    private float mPreviousX;                                 //上次的触控位置的 X 坐标
    public MySurfaceView(Context context) {/*代码省略*/}
    public boolean onTouchEvent(MotionEvent e) {/*代码省略*/} //触控事件回调方法
        private class SceneRenderer implements GLSurfaceView.Renderer {
            float yAngle;                                     //绕 Y 轴旋转的角度
            float xAngle;                                     //绕 X 轴旋转的角度
            LoadedObjectVertexOnly lovo;                   //从指定的 obj 文件中加载的对象
            public void onDrawFrame(GL10 gl) {
                GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT//清除深度缓冲与颜色缓冲
                | GLES20.GL_COLOR_BUFFER_BIT);
                MatrixState.pushMatrix();                  //保护现场
                MatrixState.translate(0, -2f, -25f);      //平移坐标系
                MatrixState.rotate(yAngle, 0, 1, 0);      //绕 Y 轴旋转
                MatrixState.rotate(xAngle, 1, 0, 0);      //绕 X 轴旋转
                if (lovo != null) {lovo.drawSelf();}      //若加载的物体不为空则绘制物体
                MatrixState.popMatrix();                   //恢复现场
            }
            public void onSurfaceChanged(GL10 gl, int width, int height) {/*代码省略*/}
            public void onSurfaceCreated(GL10 gl, EGLConfig config) {
                GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);//设置屏幕背景色 RGBA
                GLES20.glEnable(GLES20.GL_DEPTH_TEST);       //打开深度检测
                GLES20.glEnable(GLES20.GL_CULL_FACE);        //打开背面剪裁
                MatrixState.setInitStack();                  //初始化变换矩阵
                lovo = LoadUtil.loadFromFileVertexOnly("ch.obj",//加载要绘制的物体
                    MySurfaceView.this.getResources(), MySurfaceView.this);
    }}}
```

- 第11-13行为场景渲染器类的成员变量，其中lovo为LoadedObjectVertexOnly对象的引用。
- 第25-32行为onSurfaceCreated方法的实现，其中在初始化变换矩阵后创建了LoadedObjectVertexOnly类的对象。
- 第14-23行为onDrawFrame方法的实现，其中在清除深度缓冲与颜色缓冲后，在适当的位置绘制了加载的物体对象，本案例中为一只茶壶。

（4）介绍完Java代码后就应该介绍着色器了，首先是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第9章/Sample9_1/com/bn/Sample9_1_/assets目录下的vertex_color.sh。

```
uniform mat4 uMVPMatrix;                          //总变换矩阵
attribute vec3 aPosition;                         //从渲染管线接收的顶点位置
varying vec3 vPosition;                           //用于传递给片元着色器的顶点位置
void main() {
  gl_Position = uMVPMatrix * vec4(aPosition,1);   //根据总变换矩阵计算此次绘制此顶点的位置
  vPosition=aPosition;                            //将顶点位置传递给片元着色器
}
```

> **提示**
>
> 此顶点着色器与一般案例中的基本没有区别，主要是增加了通过易变变量vPosition将顶点的位置传递给片元着色器的代码。

（5）接着需要介绍的是片元着色器，其代码如下。

代码位置：见随书光盘中源代码/第9章/Sample9_1/com/bn/Sample9_1_/assets目录下的frag_color.sh。

```
precision mediump float;                              //给出默认浮点精度
varying vec3 vPosition;                               //从片元着色器接收的顶点位置
void main() {
  vec4 bColor=vec4(0.678,0.231,0.129,0);              //条纹的颜色(深红色)
  vec4 mColor=vec4(0.763,0.657,0.614,0);              //间隔区域的颜色(淡红色)
  float y=vPosition.y;                                //提取顶点的 y 坐标值
  float qjz=mod((y+100.0)*4.0,4.0);                   //折算出区间值
  if(qjz >1.8) {                                      //当区间值大于指定值时
    gl_FragColor = bColor;                            //设置片元颜色为条纹的颜色
      } else {                                        //当区间值不大于指定值时
        gl_FragColor = mColor;                        //设置片元颜色为间隔区域的颜色
  }}
```

提示

上述片元着色器主要是实现了条纹着色的效果（如图9-5所示），实现策略为根据片元的Y坐标所处的位置来决定片元是采用条纹颜色（深红色）还是间隔颜色（淡红色）进行着色。

9.2.2　加载后自动计算面法向量

前面小节案例中加载的茶壶仅采用了条纹着色策略，看起来效果不是很好，立体感不强。这是因为现实世界中的物体都是有光照的，有了光照以后层次感、立体感都会增加很多。因此本小节将给出一个根据加载的顶点及三角形面的数据自动计算法向量并施加光照的案例，其运行效果如图9-6所示。



图9-6　案例Sample9_2的运行效果图

说明

从图9-6中可以看出，添加了光照效果后，茶壶的立体感增加了很多。因此实际开发中只要条件允许，一般都会采用光照效果。

了解了本案例的运行效果后，下面就可以进行案例的开发了。由于实际上本案例仅仅是将上一小节的案例Sample9_1复制了一份并进行了修改，因此这里仅给出修改的步骤，具体如下所列。

（1）首先将加载后用于渲染物体的LoadedObjectVertexOnly类名称改为LoadedObjectVertexNormal，并在其中增加服务于光照的相关代码，具体内容如下。

代码位置：见随书光盘中源代码/第9章/Sample9_2/com/bn/Sample9_2目录下的LoadedObjectVertexNormal.java。

```
package com.bn.Sample9_2;                                   //声明包
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
public class LoadedObjectVertexNormal {//加载后的物体——携带顶点信息，自动计算面法向量
    ……//此处省略了部分成员变量的声明，读者可自行查看随书光盘的源代码
    FloatBuffer mNormalBuffer;                              //顶点法向量数据缓冲
    int vCount = 0;                                         //顶点数量
  public LoadedObjectVertexNormal(MySurfaceView mv, float[] vertices,float[] normals)
{
        initVertexData(vertices, normals);                  //初始化顶点坐标与着色数据
        initShader(mv);                                     //初始化着色器
        }
        public void initVertexData(float[] vertices, float[] normals) {
                                                            //初始化顶点数据的方法
            ……//此处省略了部分代码，与前面案例中的相同
            ByteBuffer cbb = ByteBuffer.allocateDirect(normals.length * 4);
                                                            //创建顶点法向量数据缓冲
            cbb.order(ByteOrder.nativeOrder());             //设置字节顺序
            mNormalBuffer = cbb.asFloatBuffer();            //转换为Float型缓冲
            mNormalBuffer.put(normals);                     //向缓冲区中放入顶点法向量数据
            mNormalBuffer.position(0);                      //设置缓冲区起始位置
        }
        ……//此处省略了initShader方法和drawSelf方法，读者可自行查看随书光盘的源代码
  }
```

提示

从上述代码中可以看出此类的代码与前面很多案例的并没有太多不同，只是法向量数据是由构造器参数中传入的，而不是在该类中初始化顶点数据的initVertexData方法中直接给出或计算出来的。

（2）接着对加载obj模型文件的工具类LoadUtil进行了修改，具体代码如下。

代码位置：见随书光盘中源代码/第9章/Sample9_2/com/bn/Sample9_2目录下的LoadUtil.java。

```
package com.bn.Sample9_2;                                     //声明包
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
public class LoadUtil {                          //用于从obj文件中加载3D模型的工具类
    public static float[] getCrossProduct(                    //求两个向量叉积的方法
            float x1, float y1, float z1,float x2, float y2, float z2) {
        float A = y1 * z2 - y2 * z1;                          //求出两个向量的叉积向量
        float B = z1 * x2 - z2 * x1;                          //在X、Y、Z轴上的分量A、B、C
        float C = x1 * y2 - x2 * y1;
        return new float[] { A, B, C };                       //返回叉积结果向量
        }
        public static float[] vectorNormal(float[] vector) { //向量规格化的方法
            float module = (float) Math.sqrt(vector[0] * vector[0] + //求向量的模
                    vector[1]* vector[1] + vector[2] * vector[2]);
            return new float[] { vector[0] / module,      //返回规格化的向量
                    vector[1] / module, vector[2] / module };
        }
        //从obj文件中加载仅携带顶点信息的物体，并计算面法向量
        public static LoadedObjectVertexNormal loadFromFile (String fname,
                Resources r, MySurfaceView mv) {
            LoadedObjectVertexNormal lo = null;               //加载后物体的引用
            ArrayList<Float> alv = new ArrayList<Float>();         //原始顶点坐标列表
            ArrayList<Float> alvResult = new ArrayList<Float>(); //结果顶点坐标列表
            ArrayList<Float> alnResult = new ArrayList<Float>(); //结果法向量列表
            try {
                ……//此处省略了打开输入流的代码，与前面的案例相同
                String temps = null;
                //循环不断从文件中读取行，根据行类型的不同执行不同的处理逻辑
                while ((temps = br.readLine()) != null) {//读取一行文本
                  String[] tempsa = temps.split("[ ]+"); //将文本行用空格符切分
                  if (tempsa[0].trim().equals("v")) {   //顶点坐标行
                    alv.add(Float.parseFloat(tempsa[1])); //将顶点的x、y、z坐标存入
                    alv.add(Float.parseFloat(tempsa[2])); //原始顶点坐标列表
                    alv.add(Float.parseFloat(tempsa[3]));
                  } else if (tempsa[0].trim().equals("f")) {//面数据行
                    int index = Integer.parseInt(tempsa[1].split("/")[0]) - 1;
                                                          //得到顶点编号
                    float x0 = alv.get(3 * index);           //将三角形第1个顶点的
                    float y0 = alv.get(3 * index + 1); //x、y、z坐标取出
                    float z0 = alv.get(3 * index + 2);
                    alvResult.add(x0); alvResult.add(y0); alvResult.add(z0);
                    ……//此处省略了提取第二和第三个三角形顶点坐标的代码
                    //求三角形中第一个点到第二个点的向量
                    float vxa=x1 - x0;float vya = y1 - y0; float vza = z1 - z0;
                    //求三角形中第一个点到第三个点的向量
                    float vxb=x2 - x0;float vyb = y2 - y0; float vzb = z2 - z0;
                    float[] vNormal =//通过求两个向量的叉积计算出此三角形面的法向量
```

```
46                      vectorNormal(getCrossProduct(vxa, vya,vza, vxb, vyb, vzb));
47                      for (int i = 0; i < 3; i++) { //将计算出的法向量添加到结果法向量列表中
48                      alnResult.add(vNormal[0]);alnResult.add(vNormal[1]);alnResult.
                        add(vNormal[2]);
51                  }}}
52                  int size = alvResult.size();   //将顶点数据存入数组
53                  float[] vXYZ = new float[size];
54                  for (int i = 0; i < size; i++) {vXYZ[i] = alvResult.get(i);}
55                  size = alnResult.size();  //将法向量数据存入数组
56                  float[] nXYZ = new float[size];
57                  for (int i = 0; i < size; i++) {nXYZ[i] = alnResult.get(i);}
58                  lo = new LoadedObjectVertexNormal(mv, vXYZ, nXYZ);//创建加载物体对象
59              } catch (Exception e) {Log.d("load error", "load error");e.printStackTrace();}
60              return lo;                     //返回创建的物体对象的引用
61      }}
```

说明

上述代码中最重要的是在第41-51行增加了计算每个三角形面的法向量并存入结果法向量列表的代码。基本的计算思路为，首先求出三角形中一个点到另外两个点的向量，然后将求出的两个向量进行叉积即可得出三角形面的法向量。

修改完Java代码后也需要对着色器进行一些调整，主要是增加了声明光照计算需要的一些属性变量与一致变量的代码，同时还增加了实施光照计算的代码。这些需要增加的代码与前面介绍光照的第6章案例中的基本一致，这里不再赘述，需要的读者可以参考随书光盘中的源代码。

9.2.3　加载后自动计算平均法向量

上一小节的案例成功为加载的物体添加了光照效果，但从图9-6中读者可能会发现茶壶的表面不是平滑的，而是由很多的小平面组成。这是因为上一小节中采用的是面法向量，而绘制平滑曲面时应该采用点平均法向量。

提示

前面的章节也介绍过，点平均法向量是指当一个顶点属于不止一个平面时，其法向量采用其所属多个平面各自法向量的平均值。采用点平均法向量后，绘制出来的物体表面就平滑了。

本小节将给出一个对加载的物体采用点平均法向量进行光照渲染的案例Sample9_3，其运行效果如图9-7所示。



图9-7　案例Sample9_3的运行效果

了解了本案例的运行效果后，下面就可以进行案例的开发了。由于实际上本案例仅仅是将上一 小节的案例Sample9_2复制了一份并进行了修改，因此这里仅给出修改的步骤，具体如下所列。

说明

从图9-7中可以看出，采用了点平均法向量后绘制出来的茶壶又逼真了很多。但读者也不要认为点平均法向量就优于面法向量，关键取决于待绘制物体表面的性质，棱角分明的物体还是需要采用面法向量进行光照渲染的。

（1）为了便于计算点平均法向量，首先需要在项目中添加一个表示顶点法向量的类——Normal，其代码如下。

代码位置：见随书光盘中源代码/第9章/Sample9_3/com/bn/Sample9_3目录下的Normal.java。

```java
package com.bn.Sample9_3;                                          //声明包
import java.util.Set;                                              //引入相关类
public class Normal {                    //表示法向量的类，此类的一个对象表示一个法向量
    public static final float DIFF = 0.0000001f;          //判断两个法向量是否相同的阈值
    float nx;float ny;float nz;                                    //法向量在 X、Y、Z 轴上的分量
    public Normal(float nx, float ny, float nz) {
        this.nx = nx; this.ny = ny; this.nz = nz;   //对法向量的 X、Y、Z 分量进行初始化
    }
    @Override
        public boolean equals(Object o) {
            //若两个法向量 X、Y、Z 3 个分量的差都小于指定的阈值则认为这两个法向量相等
            if (o instanceof Normal) {
                Normal tn = (Normal) o;
                if (Math.abs(nx - tn.nx) < DIFF && Math.abs(ny - tn.ny) < DIFF
                        && Math.abs(ny - tn.ny) < DIFF) {
                    return true;
                } else {return false;}
            } else {return false;}
        }
        @Override
        public int hashCode() {    //由于要用到 HashSet，因此一定要重写 hashCode 方法
            return 1;
        }
        public static float[] getAverage(Set<Normal> sn) {   //求法向量平均值的工具方法
            float[] result = new float[3];      //存放法向量 X、Y、Z 分量和的数组
            for (Normal n : sn) {               //把集合中所有的法向量 X、Y、Z 分量各自求和
                result[0] += n.nx;result[1] += n.ny;result[2] += n.nz;
            }
            return LoadUtil.vectorNormal(result);//将求和后的法向量规格化以得到平均法向量
    }}
```

- 第10-19行为重写的equals方法，其功能为比较两个法向量是否相等。需要特别注意的是，由于浮点数有计算误差，事实上相等的法向量其浮点值也很难做到绝对一致。因此比较时需要设定一个阈值，当差小于阈值时就认为相等，笔者采用的阈值为0.0000001。
- 第24-30行为求一组法向量平均值的工具方法getAverage，其入口参数是一个法向量集合。求一组法向量平均值的策略为，首先将所有法向量的X、Y、Z分量值各自求和得到和向量，然后将和向量规格化即可得到平均法向量。

（2）接着对创建加载后物体的工具类LoadUtil进行了修改，使其可以产生顶点的平均法向量数据，具体代码如下。

代码位置：见随书光盘中源代码/第9章/Sample9_3/com/bn/Sample9_3目录下的LoadUtil.java。

```
package com.bn.Sample9_3;                                      //声明包
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
public class LoadUtil {
 ……//此处省略了 getCrossProduct 和 vectorNormal 方法，读者可自行查看随书光盘的源代码
 public static LoadedObjectVertexNormal loadFromFile(//从 obj 文件中加载携带顶点信息的物体，
  String fname,Resources r, MySurfaceView mv) {      //并自动计算每个顶点的平均法向量
    LoadedObjectVertexNormal lo = null;              //加载后物体的引用
    ArrayList<Float> alv = new ArrayList<Float>();  //原始顶点坐标列表
    ArrayList<Integer> alFaceIndex = new ArrayList<Integer>();   //结果顶点编号列表
        ArrayList<Float> alvResult = new ArrayList<Float>();     //结果顶点坐标列表
        //平均前各个编号的点的法向量集合的 Map，此 HashMap 的 key 为点的编号，
        //value 为点所在的各个面的法向量的集合
        HashMap<Integer, HashSet<Normal>> hmn = new HashMap<Integer, HashSet<Normal>>();
        try { ……//此处省略了打开输入流的代码，与前面的案例相同
            String temps = null;
            //扫描文件，根据行类型的不同执行不同的处理逻辑
            while ((temps = br.readLine()) != null) {        //读取一行文本
            String[] tempsa = temps.split("[ ]+");           //将文本行用空格符切分
            if (tempsa[0].trim().equals("v")) {              //顶点坐标行
                ……//此处省略顶点坐标行的处理代码，与上一个案例相同
            } else if (tempsa[0].trim().equals("f")) {  //面数据行
              int[] index = new int[3];                    //此面中 3 个顶点编号值的数组
              index[0] = Integer.parseInt(tempsa[1].split("/")[0]) - 1;//得到顶点编号
              float x0 = alv.get(3 * index[0]);            //将三角形第 1 个顶点的
              float y0 = alv.get(3 * index[0] + 1);        //x、y、z 坐标取出
              float z0 = alv.get(3 * index[0] + 2);
              alvResult.add(x0);alvResult.add(y0);alvResult.add(z0);
              ……//此处省略了提取三角形第二和第三个顶点坐标的代码
              alFaceIndex.add(index[0]);                   //记录此三角形面 3 个顶点的索引
              alFaceIndex.add(index[1]);
              alFaceIndex.add(index[2]);
              //求三角形中第一个点到第二个点的向量
              float vxa = x1 - x0;float vya = y1 - y0;float vza = z1 - z0;
              //求三角形中第一个点到第三个点的向量
              float vxb = x2 - x0;float vyb = y2 - y0;float vzb = z2 - z0;
              float[] vNormal = //通过求两个向量的叉积计算法出此三角形面的法向量
               vectorNormal(getCrossProduct(vxa, vya, vza, vxb, vyb,vzb));
              //将此三角形面的法向量记录到此面 3 个顶点各自的法向量集合中
              for (int tempInxex : index) {
                HashSet<Normal> hsn = hmn.get(tempInxex); //获取指定编号顶点的法向量集合
                if (hsn == null) {hsn=new HashSet<Normal>();}//若集合不存在则创建
                //将此点的法向量添加到集合中
                hsn.add(new Normal(vNormal[0], vNormal[1], vNormal[2]));
                hmn.put(tempInxex, hsn);         //将法向量集合放进 HsahMap 中
```

```
            }}}
        ....//此处省略了将顶点数据存入数组的代码，与上一个案例的相同
        float[] nXYZ = new float[alFaceIndex.size() * 3];//用于存放法向量数据的数组
          int c = 0;                                 //法向量数组索引
          for (Integer i : alFaceIndex) {            //对结果序列中每个顶点的编号循环
              //根据当前的编号从 Map 中取出一个顶点的法向量的集合
              HashSet<Normal> hsn = hmn.get(i);
              float[] tn = Normal.getAverage(hsn);//求出集合中法向量的平均法向量
              //将计算出的平均法向量存放到法向量数组中
              nXYZ[c++] = tn[0];nXYZ[c++] = tn[1];nXYZ[c++] = tn[2];
          }
          lo = new LoadedObjectVertexNormal(mv, vXYZ, nXYZ);//创建加载物体对象
      } catch (Exception e) {Log.d("load error", "load error"); e.printStackTrace();}
      return lo;                                     //返回创建的物体对象的引用
}}
```

说明

上述代码中在计算出各个面的法向量后不是直接送入结果法向量列表，而是将法向量记录进各个顶点对应的法向量集合中。当所有面的法向量计算结束后，再求出各个顶点的平均法向量，接着送入法向量数据数组以供创建加载物体对象。

由于本案例中用到的着色器与上一个案例的完全相同，因此这里不再赘述。

9.2.4　加载纹理坐标

上一小节已经给出了对加载后的物体采用点平均法向量实施光照的案例，效果已经很不错。但现实世界中的物体其表面并不一定是纯色的，可能是有花纹的，如青花瓷的茶壶。本小节将给出加载时也加载纹理坐标信息的案例Sample9_4，其运行效果如图9-8所示。



图9-8　案例Sample9_4的运行效果

> **说明**
>
> 从图9-8中可以看出，在光照的基础上又增加了纹理映射后茶壶更漂亮了。另外，由于本书插图是灰度印刷的，因此建议读者自行运行本案例进行观察或参考本书前面的彩图。

了解了本案例的运行效果后，下面就可以进行案例的开发了。由于实际上本案例仅仅是将上一小节的案例Sample9_3复制了一份并进行了修改，因此这里仅给出修改的步骤，具体如下所列。

（1）首先将加载后用于渲染物体的LoadedObjectVertexNormal类名称改为LoadedObjectVertex NormalTexture，并在其中增加了加载纹理坐标数据的相关代码，具体内容如下。

代码位置：见随书光盘中源代码/第9章/Sample9_4/com/bn/Sample9_4目录下的LoadedObjectVertexNormalTexture.java。

```
package com.bn.Sample9_4;                                //声明包
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
public class LoadedObjectVertexNormalTexture {
    ……//此处省略了部分成员变量，读者可自行查看随书光盘的源代码
    FloatBuffer mTexCoorBuffer;                          //顶点纹理坐标数据缓冲
    public LoadedObjectVertexNormalTexture(MySurfaceView mv,
          float[] vertices,float[] normals, float texCoors[]) {
        initVertexData(vertices, normals, texCoors);     //初始化顶点坐标与着色数据
        initShader(mv);                                  //初始化着色器
        }
        public void initVertexData(                      //初始化顶点数据的方法
              float[] vertices, float[] normals, float texCoors[]) {
          ……//此处省略了顶点坐标和法向量数据初始化的代码，读者可自行查看随书光盘的源代码
          ByteBuffer tbb = ByteBuffer.allocateDirect(texCoors.length * 4);
          tbb.order(ByteOrder.nativeOrder());            //设置字节顺序
          mTexCoorBuffer = tbb.asFloatBuffer();          //转换为 Float 型缓冲
          mTexCoorBuffer.put(texCoors);                  //向缓冲区中放入顶点纹理坐标数据
          mTexCoorBuffer.position(0);                    //设置缓冲区起始位置
        }
        ……//此处省略了 initShader 方法，读者可自行查看随书光盘的源代码
        public void drawSelf(int texId) {                //绘制加载物体的方法
          ……//此处省略了部分代码，与前面案例中的相同
           GLES20.glVertexAttribPointer (maTexCoorHandle,2,//将顶点纹理坐标数据传入渲染管线
            GLES20.GL_FLOAT,false, mTexCoorBuffer);
            ……//此处省略了部分代码，与前面案例中的相同
            GLES20.glEnableVertexAttribArray(maTexCoorHandle);     //启用纹理坐标数据
            GLES20.glActiveTexture(GLES20.GL_TEXTURE0);            //启用 0 号纹理
            GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, texId);     //绑定纹理
            GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount);  //绘制加载的物体
       }
    }
```

提示

该类中主要是增加了接收纹理坐标数据，并将纹理坐标数据送入缓冲以备渲染管线使用，以及绘制时将纹理坐标数据送入渲染管线与绑定纹理的相关代码，其他部分与上一案例中的基本相同。同时需要注意的是，drawSelf绘制方法也增加了接收纹理id的入口参数texId，这样在绘制时可以根据需要传入不同的纹理，增加了程序的灵活性。

（2）接着对创建加载后物体的工具类LoadUtil进行了修改，使其可以从obj模型文件中加载纹理坐标数据，具体代码如下。

代码位置：见随书光盘中源代码/第9章/Sample9_4/com/bn/Sample9_4目录下的LoadUtil.java。

```
package com.bn.Sample9_4;                                   //声明包
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
```

```java
public class LoadUtil {
    ……//此处省略了 getCrossProduct 和 vectorNormal 方法，读者可自行查看随书光盘的源代码
    public static LoadedObjectVertexNormalTexture loadFromFile (
        String fname, Resources r, MySurfaceView mv) {
        ……//此处省略了一些成员变量的声明，与前面案例中的相同
        ArrayList<Float> alt = new ArrayList<Float>();                //原始纹理坐标列表
        ArrayList<Float> altResult = new ArrayList<Float>();          //结果纹理坐标列表
            try {……//此处省略了打开输入流的代码，与前面的案例相同
                String temps = null;
                //扫描文件，根据行类型的不同执行不同的处理逻辑
                while ((temps = br.readLine()) != null) {             //读取一行文本
                    String[] tempsa = temps.split("[ ]+");            //将文本行用空格符切分
                    if (tempsa[0].trim().equals("v")) {               //顶点坐标行
                        ……//此处省略了部分与前面案例相同的代码
                    } else if (tempsa[0].trim().equals("vt")) { //纹理坐标行
                        alt.add(Float.parseFloat(tempsa[1]) / 2.0f);//提取出 S 纹理坐标
                        alt.add(Float.parseFloat(tempsa[2]) / 2.0f); //提取出 T 纹理坐标
                    } else if (tempsa[0].trim().equals("f")) {  //面数据行
                     ……//此处省略了部分与前面案例相同的代码
                     //将三角形 3 个顶点的纹理坐标数据组织到结果纹理坐标列表中
                     int indexTex = Integer.parseInt(tempsa[1].split("/")[1]) - 1;
                                                                      //获取纹理坐标编号
                     altResult.add(alt.get(indexTex * 2));            //第 0 个顶点的纹理坐标
                     altResult.add(alt.get(indexTex * 2 + 1));
                     indexTex = Integer.parseInt(tempsa[2].split("/")[1]) - 1;
                                                                      //获取纹理坐标编号
                     altResult.add(alt.get(indexTex * 2));            //第 1 个顶点的纹理坐标
                     altResult.add(alt.get(indexTex * 2 + 1));
                     indexTex = Integer.parseInt(tempsa[3].split("/")[1]) - 1;
                                                                      //获取纹理坐标编号
                     altResult.add(alt.get(indexTex * 2));            //第 2 个顶点的纹理坐标
                     altResult.add(alt.get(indexTex * 2 + 1));
                }}
                ……//此处省略了部分与前面案例相同的代码
                size = altResult.size();
                float[] tST = new float[size];      //用于存放结果纹理坐标数据的数组
                for (int i = 0;i<size;i++){tST[i]=altResult.get(i);}
                                                    //将纹理坐标数据存入数组
                //创建加载物体对象
                lo = new LoadedObjectVertexNormalTexture(mv, vXYZ, nXYZ, tST);
            } catch (Exception e) {Log.d("load error", "load error");e.printStackTrace();}
            return lo;                              //返回创建的物体对象的引用
    }}
```

说明

上述代码中主要是增加了从obj文件中加载顶点纹理坐标数据，并将纹理坐标数据存放到数组中供创建加载对象使用的相关代码，其他的部分与前面的案例基本相同。另外要注意，本案例中在加载纹理坐标时将加载后的值除以了2，这是为了减少纹理的重复次数，读者可以根据具体情况决定。

（3）完成了LoadUtil类的修改后，就需要对MySurfaceView类进行修改了，具体代码如下。

代码位置：见随书光盘中源代码/第9章/Sample9_4/com/bn/Sample9_4目录下的MySurfaceView.java。

```java
package com.bn.Sample9_4;                                    //声明包
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
class MySurfaceView extends GLSurfaceView {
    ……//此处省略了部分成员变量的声明，与前面案例中的相同
    int textureId;                                          //系统分配的纹理id
    public MySurfaceView(Context context) {/*代码省略*/}
    public boolean onTouchEvent(MotionEvent e) {/*代码省略*/}
    private class SceneRenderer implements GLSurfaceView.Renderer {
        float yAngle; float xAngle;                         //绕Y轴、X轴旋转的角度
            LoadedObjectVertexNormalTexture lovo;           //从指定的obj文件中加载的对象
            public void onDrawFrame(GL10 gl) {
                ……//此处省略了部分代码，与前面案例中的相同
                if (lovo != null) {lovo.drawSelf(textureId);}//若加载的物体不为空则绘制物体
                ……//此处省略了部分代码，与前面案例中的相同
            }
            public void onSurfaceChanged(GL10 gl, int width, int height) {/*代码省略
*/}
            public void onSurfaceCreated(GL10 gl, EGLConfig config) {
                ……//此处省略了部分代码，与前面案例中的相同
                textureId = initTexture(R.drawable.ghxp);       //加载纹理图
        }}
        public int initTexture(int drawableId){/*代码省略*/}
    }
```

说明

上述代码中主要是增加了加载纹理需要的相关代码，包括纹理id成员变量textureId的声明、加载纹理的initTexture方法，以及

调用initTexture方法加载纹理的代码。其中加载纹理的initTexture方法与第7章案例中的完全相同，这里不再赘述。

本案例中用到的着色器主要是在上一个案例中着色器的基础上增加了接收纹理坐标、纹理数据以及进行纹理采样的相关代码，其他的部分没有变化。由于接收纹理坐标、纹理数据以及进行纹理采样的代码在第7章的案例中已经进行了详细介绍，故这里不再赘述，需要的读者请参考随书光盘中的源代码。

9.3　本章小结

本章主要介绍了如何利用3DMax软件来导出obj模型文件，以及如何在应用程序中加载模型进行渲染。有了加载模型的能力后，读者的开发能力应该有了质的提升，可以开发出任意几何形状的3D物体了，这也为后面制作复杂的3D场景打下了良好的基础。

第10章

混合与雾

前面的章节介绍了很多OpenGL ES 2.0开发的基础知识，掌握了这些基础知识已经具备了搭建一些简单场景的能力。本章将在这个基础上更上一层楼，向读者介绍两种独特的场景渲染技术，混合与雾。

10.1　混合技术

到目前为止，本书案例场景中的物体都是不透明的，这在很多情况下已经能够满足需求。但现实世界中还有很多半透明的物体，如果希望在场景中真实再现此类物体，最常用的技术就是混合。本节将详细介绍混合各方面的知识，最后再给出两个使用混合的小案例。

10.1.1　混合基本知识

顾名思义，混合技术就是将两个片元调和，主要用于将通过各项测试准备进入帧缓冲的片元（源片元）与原有片元（目标片元）按照设定的比例加权计算出最终片元的颜色值。也就是说在启用了混合的情况下，新片元将不再是直接覆盖缓冲区中的原片元，具体情况如图10-1所示。

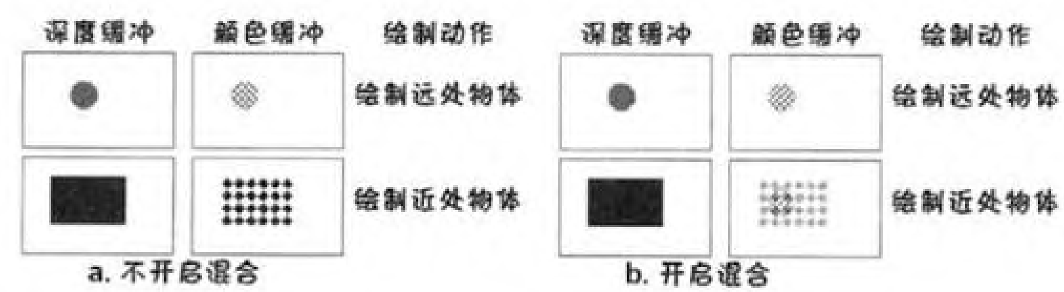


图10-1　通过混合实现半透明原理

> **说明**
>
> 从图10-1中可以看出，开启了混合以后，最终看到的场景是近处的物体透出了一些远处物体的内容。当然透过比例的多少取决于参数的设置，后面将详细介绍。另外，图10-1深度缓冲中颜色越深表示深度越小，这是因为从灰度值的角度考虑值越大颜色越浅。

从前面的介绍中已经知道，混合前需要首先设定加权比例。OpenGL ES 2.0中是通过设置混合因子来指定两个片元的加权比例的，每次都需要给出两个混合因子，具体如下所列。

- 第一个是源因子，其用于确定将进入帧缓冲的片元在最终片元中的比例。
- 第二个是目标因子，其用于确定原帧缓冲中的片元在最终片元中的比例。

由于OpenGL ES 2.0中的每个颜色值包括4个色彩通道，因此两种因子都有4个分量值，分别对应一个色彩通道，具体的混合计算细节如下所列。

- 设源因子和目标因子分别为[S_r ，S_g ，S_b ，S_a]和[D_r ，D_g ，D_b ，D_a]，S表示是源因子，D表示是目标因子，r、g、b、a下标分别表示红、绿、蓝、透明度4个色彩通道。
- 设源片元与目标片元的颜色值分别为[R_s ，G_s ，B_s ，A_s]和[R_d ，G_d ，B_d ，A_d]，R、G、B、A分别表示红、绿、蓝、透明度4个色彩通道，s下标表示源片元，d下标表示目标片元。
- 混合后最终片元颜色各个色彩通道的值为[$R_s S_r + R_d D_r$ ，$G_s S_g + G_d D_g$ ，$B_s S_b + B_d D_b$ ，$A_s S_a + A_d D_a$]。

提示

经过加权计算后，最终片元的某些通道值可能会超过1.0，此时渲染管线会自动执行截取操作，将大于1.0的通道值设置为1.0。

10.1.2　源因子和目标因子

从上一小节的介绍中可以看出，运用混合技术最重要的就是设置合适的混合因子。在其他一些3D渲染平台中，混合因子4个通道的值可以由开发人员自由设置，如OpenGL、DirectX等。但OpenGL ES处于简化的考虑，不允许开发人员任意设置混合因子的值，只允许开发人员根据需要从系统预置的因子值（GLES20类的静态常量）中选取，常用的混合因子值如表10-1所列。

表10-1　系统提供的源因子和目标因子

常量名	RGB混合因子	A混合因子
GL_ZERO	[0，0，0]	0
GL_ONE	[1，1，1]	1
GL_SRC_COLOR	[R_s ，G_s ，B_s]	A_s
GL_ONE_MINUS_SRC_COLOR	[$1-R_s$ ，$1-G_s$ ，$1-B_s$]	$1-A_s$

GL_DST_COLOR	[R_d , G_d , B_d]	A_d
GL_ONE_MINUS_DST_COLOR	[$1-R_d$, $1-G_d$, $1-B_d$]	$1-A_d$
GL_SRC_ALPHA	[A_s , A_s , A_s]	A_s
GL_ONE_MINUS_SRC_ALPHA	[$1-A_s$, $1-A_s$, $1-A_s$]	$1-A_s$
GL_DST_ALPHA	[A_d , A_d , A_d]	A_d
GL_ONE_MINUS_DST_ALPHA	[$1-A_d$, $1-A_d$, $1-A_d$]	$1-A_d$
GL_SRC_ALPHA_SATURATE	[f, f, f] f=min（A_s , $1-A_d$ ）	1

提示

常量名称中有SRC的代表各通道值来自源片元，有DST的代表各通道值来自目标片元，另外GL_SRC_ALPHA_SATURATE只能用作源因子。

表10-1　中每行右侧的两个列给出了此行混合因子4个通道的值，执行混合时，渲染管线将采 用这些值依照上一小节给出的计算方法进行计算。读者可以根据混合的需求选择不同的源因子与目标因子组合，恰当的组合可以产生很好的效果。

下面给出两种常用的组合，具体情况如下所列。

- 源因子为GL_SRC_ALPHA，目标因子为GL_ONE_MINUS_SRC_ALPHA，即源因子和目标因子分别为[As，As, As, As]和[1-As，1-As, 1-As, 1-As]。此组合实现的是最典型的半透明遮挡效果；若源片元是透明的，则根据透明度透过后面的内容；若源片元不透明，则仅能看到源片元。因此，使用此组合时往往会采用半透明的纹理或颜色对源片元着色。
- 源因子为GL_SRC_COLOR，目标因子为GL_ONE_MINUS_SRC_COLOR，即源因子和目标因子分别为[Rs，Gs, Bs, As]和[1-Rs, 1-Gs，1-Bs, 1-As]。此组合可以实现滤光镜效果，也就是平时透过有色眼镜或玻璃窗观察事物的感觉。与第一种常用组合不同，此组合不要求应用于源片元的颜色或纹理是半透明的。

10.1.3 简单混合效果案例

上一小节中介绍了两种常用的混合因子组合，本小节将通过两个简单的案例来说明这两种混合因子组合的使用。首先介绍的是采用滤光镜效果因子组合的案例Sample10_1，其运行效果如图10-2所示。

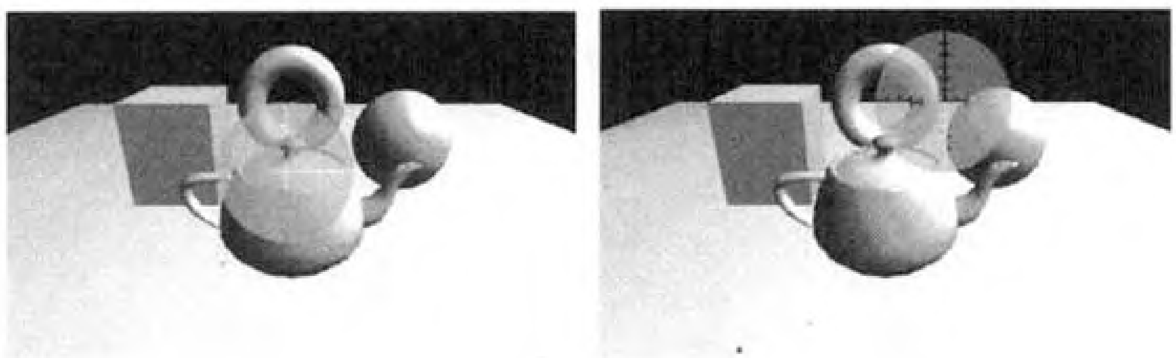


图10-2 案例Sample10_1的运行效果图

提示

从图10-2中可以看出，场景中有一个可以移动的类似枪瞄镜的圆形，透过圆形可以看到后面的物体。本案例中的圆形实际是绿色的，由于灰度印刷的原因读者可能看得不是很清楚，此时请读者自行运行本案例进行观察。另外，案例运行时手指触摸屏幕的左右两侧，滤光镜圆形会左右移动，手指触摸屏幕中间的上下两侧，滤光镜会上下移动，如图10-3所示。

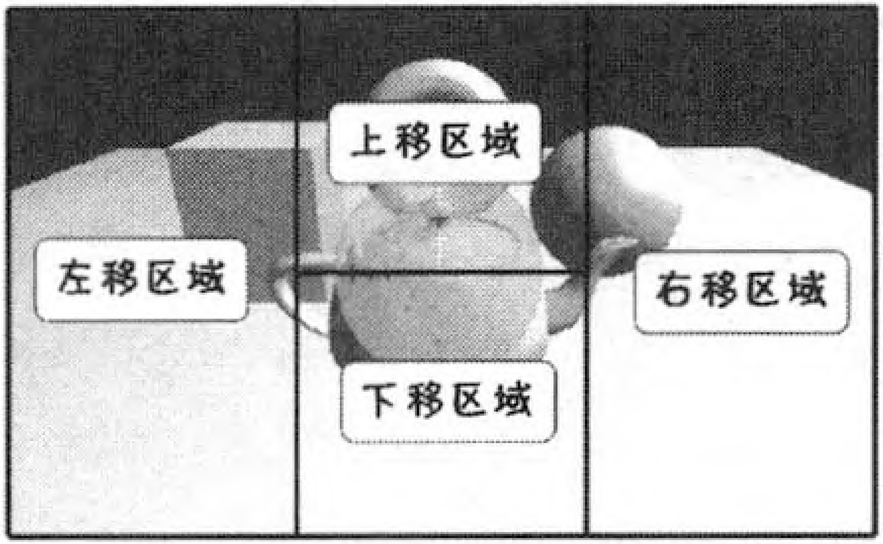


图10-3　触控区域

了解了案例的运行效果及操控方式后就可以介绍案例的开发步骤了，具体如下所列。

（1）首先用3DMax生成5个基本物体（平面、圆环、茶壶、立方体、圆球），并导出成obj文件放入项目的assets目录待用。

（2）开发出搭建场景的基本代码，包括：加载物体、摆放物体、计算光照等。这些代码与前面章节的许多案例基本套路完全一致，因此这里不再赘述。

（3）开发一个纹理矩形类TextureRect，用来呈现滤光镜。此类在前面很多案例中已经出现过，这里也不再赘述。

（4）准备好本案例中需要用到的滤光镜纹理图片（lgq.png），其内容如图10-4所示。

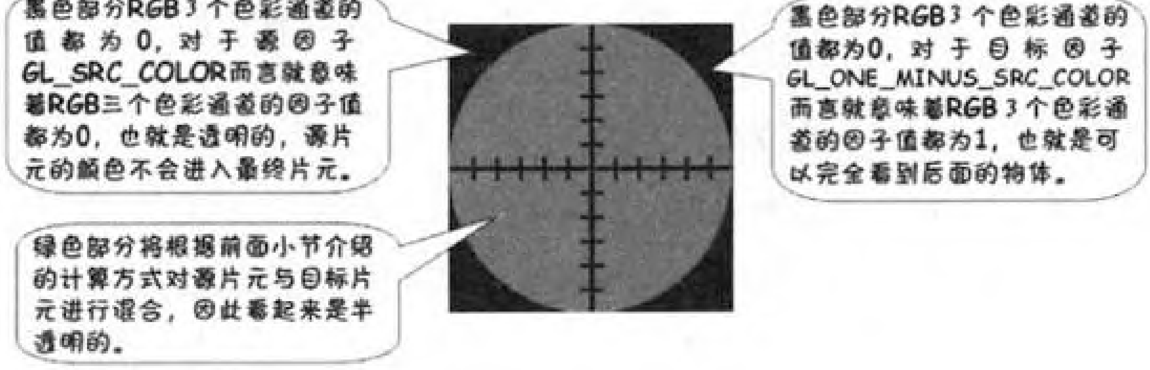


图10-4　本案例的滤光镜纹理图

（5）在渲染场景的MySurfaceView类的onDrawFrame方法中添加启用混合模式并绘制滤光镜的代码，具体内容如下。

代码位置：见随书光盘中源代码/第10章/Sample10_1/com/bn/Sample10_1目录下的MySurfaceView.java。

```
public void onDrawFrame(GL10 gl) {
    GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT        //清除深度缓冲与颜色缓冲
            | GLES20.GL_COLOR_BUFFER_BIT);
    ……//此处省略了绘制场景中物体的代码，读者可自行查看随书光盘中的源代码
    GLES20.glEnable(GLES20.GL_BLEND);                 //开启混合
    GLES20.glBlendFunc(GLES20.GL_SRC_COLOR,           //设置混合因子，其中第一个为源
            GLES20.GL_ONE_MINUS_SRC_COLOR);           //因子，第二个为目标因子
    MatrixState.pushMatrix();                         //保护现场
    MatrixState.translate(rectX, rectY, 25f);         //平移纹理矩形
        rect.drawSelf(rectTexId);                     //绘制滤光镜纹理矩形
        MatrixState.popMatrix();                      //恢复现场
        GLES20.glDisable(GLES20.GL_BLEND);            //关闭混合
        MatrixState.popMatrix();                      //恢复现场
    }
```

说明

上述代码中在绘制滤光镜时首先启用了混合，然后设置了前一小节中介绍的滤光镜因子组合，最后绘制了滤光镜纹理矩形并关闭了混合。

想实现半透明的滤光镜效果不但可以采用滤光镜因子组合，还可以采用上一小节介绍的第一种常见因子组合。下面将案例Sample10_1

复制，并进行简单修改得到采用第一种常见因子组合的案例Sample10_1a，其运行效果如图10-5所示。

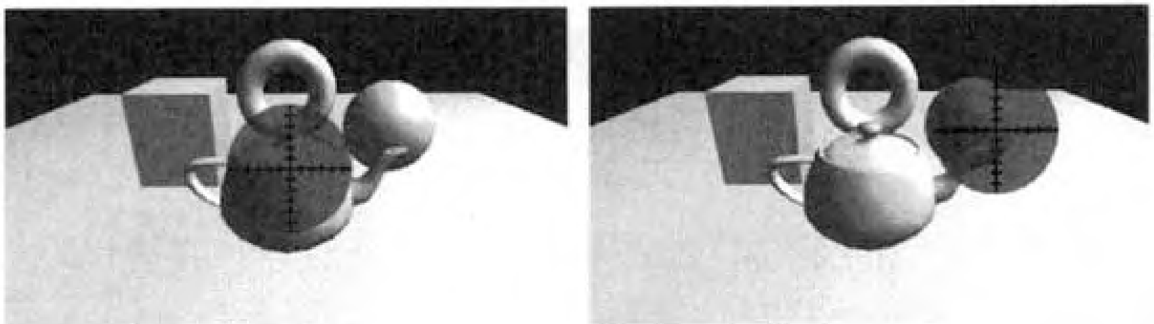


图10-5　Sample10_1a运行效果

从图10-5中可以看出，采用第一种混合因子组合也可以产生滤光镜的效果。下面简要介绍一下案例中需要修改的部分，具体如下所列。

（1）将原来黑色背景的纹理图改为透明背景，同时将绿色的瞄准镜圆形设置为半透明，改动后的纹理图如图10-6所示。

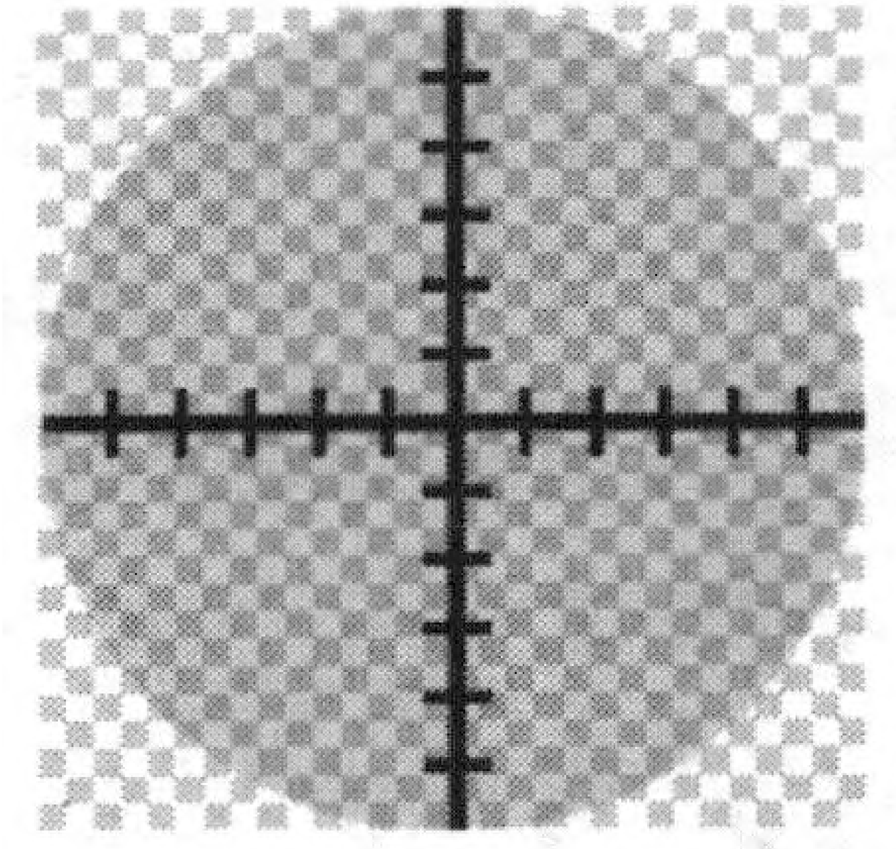


图10-6　修改后的半透明纹理

> **提示**
>
> 图10-6中灰白相间的格子表示透明背景，这种表示方式是约定俗成的。

（2）将MySurfaceView类中的混合因子修改为第一种组合，具体代码如下。

代码位置：见随书光盘中源代码/第10章/Sample10_1a/com/bn/Sample10_1a目录下的MySurfaceView.java。

```
1 public void onDrawFrame(GL10 gl) {
2     GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT          //清除深度缓冲与颜色缓冲
3             | GLES20.GL_COLOR_BUFFER_BIT);
4     ……//此处省略了绘制场景中物体的代码，读者可自行查看随书光盘中的源代码
5     GLES20.glEnable(GLES20.GL_BLEND);                  //开启混合
6     GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA,            //设置混合因子，其中第一个为源
7             GLES20.GL_ONE_MINUS_SRC_ALPHA);            //因子，第二个为目标因子
8     MatrixState.pushMatrix();                          //保护现场
9     MatrixState.translate(rectX, rectY, 25f);          //平移纹理矩形
10    rect.drawSelf(rectTexId);                          //绘制滤光镜纹理矩形
11    MatrixState.popMatrix();                           //恢复现场
12    GLES20.glDisable(GLES20.GL_BLEND);                 //关闭混合
13    MatrixState.popMatrix();                           //恢复现场
14  }
```

说明

主要是修改了代码的第6-7行中的源因子与目标因子，将原来的GL_SRC_COLOR与GL_ONE_MINUS_SRC_COLOR修改为GL_SRC_ALPHA与GL_ONE_MINUS_SRC_ALPHA。

本小节仅仅通过案例介绍了两种常用混合因子组合的使用，有兴趣的读者还可以尝试其他的一些组合，可能会产生意想不到的效果。

10.2　地月系云层效果的实现

第7章最后曾经介绍过一个地月系场景的案例Sample7_4，本小节将使用混合技术对其进行升级（升级后的案例为Sample10_2），为地

月系场景中的地球添加云层。升级后案例的运行效果如图10-7所示。

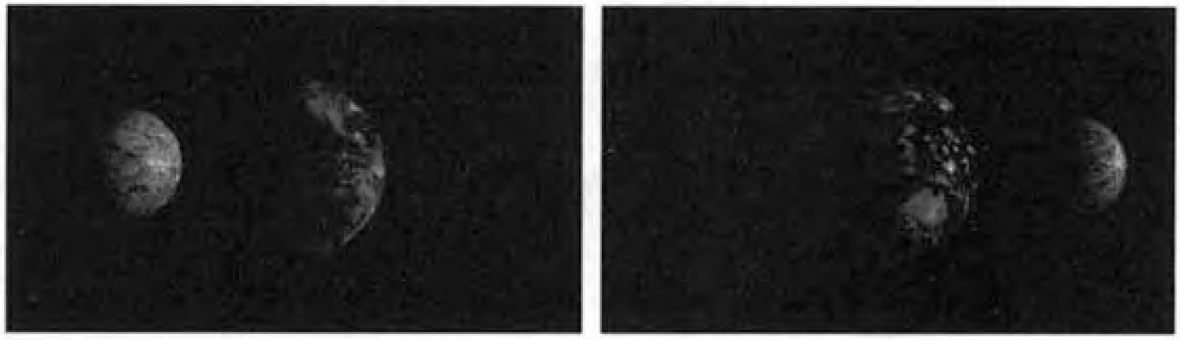


图10-7　案例Sample10_2的运行效果图

从图10-7中可以看出，添加了云层后地球更加真实了。下面请读者首先了解一下本案例中云层的纹理图cloud.jpg，其内容如图10-8所示。



图10-8　云层纹理图片

从图10-8中可以看出，此纹理图并不是透明的，因此读者一定以为本案例将采用第二种因子组合。其实不然，由于本书介绍的是功能强大的OpenGL ES 2.0，因此本案例还可以继续采用第一种因子组合，不过需要在片元着色器中根据纹理采样值的灰度设置片元的透明度。

了解了云层纹理图的基本使用策略和案例的运行效果后，就可以进行案例的开发了。由于实际上本案例仅仅是将案例Sample7_4复制了一份并进行了修改，因此这里仅给出修改的主要步骤，具体如下所列。

（1）首先在案例中加入一个新类Cloud，此类实际上是一个纹理球，与地球、月球没太大区别，主要的不同之处是采用了一套特殊的着色器。

提示

由于Cloud类与月球类代码基本相同，故这里不再赘述，需要的读者请自行参考随书光盘。

（2）接着在MySurfaceView类中增加创建云层对象和绘制云层的代码，具体内容如下。

代码位置：见随书光盘中源代码/第10章/Sample10_2/com/bn/Sample10_2目录下的MySurfaceView.java。

```
package com.bn.Sample10_2;                                          //声明包
……//此处省略了部分类的引入代码，读者可自行查看随书光盘的源代码
class MySurfaceView extends GLSurfaceView {
    private final float TOUCH_SCALE_FACTOR = 180.0f / 320;       //角度换算比例
    ……//此处省略了部分成员变量，读者可自行查看随书光盘的源代码
    int textureIdEarth;                                             //地球白天纹理id
    int textureIdEarthNight;                                        //地球夜晚纹理id
    int textureIdMoon;                                              //月球纹理id
    int textureIdCloud;                                             //云层纹理id
    public MySurfaceView(Context context) {/*代码省略*/}
    public boolean onTouchEvent(MotionEvent e) {/*代码省略*/} //触控事件回调方法
    private class SceneRenderer implements GLSurfaceView.Renderer {
        Earth earth;                                                //地球
        Moon moon;                                                  //月球
        Celestial cSmall;                                           //小星星天球
        Celestial cBig;                                             //大星星天球
        Cloud cloud;                                                //云层
```

```
        public void onDrawFrame(GL10 gl) {                 //绘制一帧画面的方法
            GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT//清除深度缓冲与颜色缓冲
                    | GLES20.GL_COLOR_BUFFER_BIT);
            MatrixState.pushMatrix();                      //保护现场
            MatrixState.rotate(eAngle, 0, 1, 0);           //地球自转
            earth.drawSelf(textureIdEarth, textureIdEarthNight);//绘制地球
            GLES20.glEnable(GLES20.GL_BLEND);              //开启混合
            GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA,        //设置混合因子
                    GLES20.GL_ONE_MINUS_SRC_ALPHA);
            cloud.drawSelf(textureIdCloud);                //绘制云层
            GLES20.glDisable(GLES20.GL_BLEND);             //关闭混合
            ……//此处省略了绘制星空和月球的代码，需要时请参考光盘
        }
        public void onSurfaceChanged(GL10 gl, int width, int height) {
          ……//此处省略了部分与修改前一致的代码，需要时请参考光盘
         textureIdCloud=initTexture(R.drawable.cloud); //加载云层纹理
         ……//此处省略了部分与修改前一致的代码，需要时请参考光盘
       }
        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
            GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);    //设置屏幕背景色RGBA
            earth = new Earth(MySurfaceView.this, 2.0f);    //创建地球对象
            cloud = new Cloud(MySurfaceView.this, 2.02f);   //创建云层对象
            moon = new Moon(MySurfaceView.this, 1.0f);      //创建月球对象
            cSmall = new Celestial(1, 0, 1000, MySurfaceView.this);//创建小星星天球对象
            cBig = new Celestial(2, 0, 500, MySurfaceView.this); //创建大星星天球对象
            GLES20.glEnable(GLES20.GL_DEPTH_TEST); //打开深度检测
            MatrixState.setInitStack();                     //初始化变换矩阵
    }}
    public int initTexture(int drawableId) {/*代码省略*/}
}
```

说明

上述代码中主要是在原来的基础上增加了加载云层纹理、创建云层对象以及绘制云层的相关代码。需要特别注意的是，绘制云层时采用的混合因子组合为GL_SRC_ALPHA，GL_ONE_MINUS_SRC_ALPHA，此组合需要源片元是透明的才有效，因此后面会在片元着色器中通过程序根据片元灰度设置片元透明度。

（3）完成了Java代码的修改后，就可以开发绘制云层的专用着色器了，首先是顶点着色器。由于绘制云层的顶点着色器与绘制月球的基本一致，因此这里不再赘述，需要的读者请参考随书光盘中源代码/第10章/Sample10_2/assets目录下的vertex_cloud.sh文件。

（4）接着需要开发的是绘制云层的片元着色器，其代码如下。

代码位置：见随书光盘中源代码/第10章/Sample10_2/assets目录下的frag.sh。

```
precision mediump float;                //给出默认的浮点精度
varying vec2 vTextureCoord;             //从顶点着色器传递过来的纹理坐标
varying vec4 ambient;                   //从顶点着色器传递过来的环境光最终强度
varying vec4 diffuse;                   //从顶点着色器传递过来的散射光最终强度
```

```
5 varying vec4 specular;                    //从顶点着色器传递过来的镜面光最终强度
6 uniform sampler2D sTexture;               //纹理内容数据
7 void main() {
8   vec4 finalColor = texture2D(sTexture, vTextureCoord);  //根据纹理坐标采样出颜色值
9   finalColor.a=(finalColor.r+finalColor.g+finalColor.b)/3.0;  //根据颜色值灰度计算透明度
10      //综合3个通道光的最终强度及片元的颜色计算出最终片元的颜色值
11      finalColor=finalColor*ambient+finalColor*specular+finalColor*diffuse;
12      gl_FragColor = finalColor;                 //将最终片元的颜色值传递给管线
13  }
```

提示

上述片元着色器代码中最大的特色就是第9行根据采样出的颜色值灰度计算出了此片元的透明度。具体方法为，将从纹理图中采样出颜色值的R、G、B通道值求平均，再将平均值作为透明度即可。由于透明度的取值范围与颜色一样都是0.0~1.0，0.0为完全透明，1.0为完全不透明，这样计算最终达到的效果就是纹理图中越黑的位置透明度就越高。

10.3　雾

前面章节中给出了不少真实场景的案例，这些案例中的物体无论远近看起来都一样清晰。这样虽然也不错，但并不完全符合现实世界的情况。现实世界中由于有大气、灰尘、雾等的影响，随着距离的加大物体将越来越不清晰，最终彻底融入背景中。

本节将向读者介绍如何通过OpenGL ES 2.0实现类似于雾的效果，主要包括雾的原理与优势、雾的简单实现等两部分内容。

10.3.1　雾的原理与优势

本小节所指的雾是一个通用术语，其不仅是用于实现现实世界中的雾，还包括用于实现烟雾和污染等大气效果。使用雾效果可以使距离摄像机较远的物体融入到雾的颜色中。而且物体离摄像机越远，雾的颜色越浓，物体看起来越不清晰。很多流行的3D游戏场景中都是用了雾效果，如非常著名的古剑奇谭，其场景效果如图10-9所示。

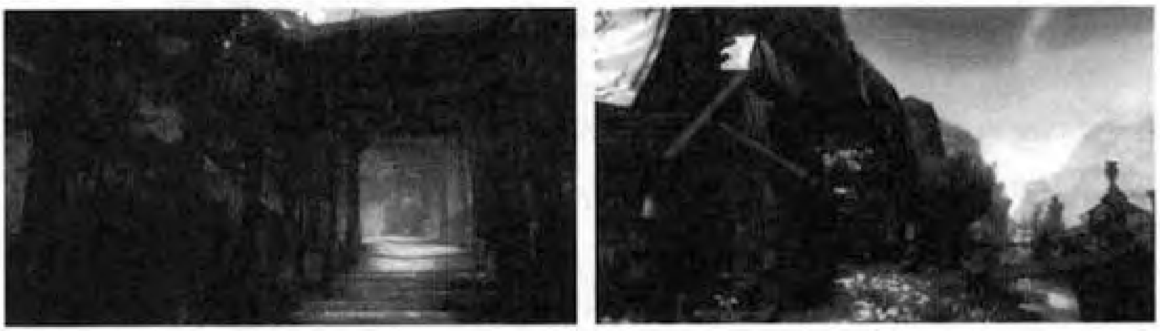


图10-9　古剑奇谭游戏场景截图

> **提示**
>
> 在场景中使用雾不但可以提高真实感，特定的情况下还能优化性能。具体是指当物体离摄像机足够远时，雾就足够浓，此时只能看到雾而看不到物体，也就不必对物体的着色进行详细计算了，这样可以大大提高渲染效率。

实现雾效果有很多的数学模型，首先介绍最为简单的线性模型，此模型的计算公式如下。

$$f = \max(\min((end- dist)/(end-start),1.0),0.0)$$

- f为雾化因子，其取值范围为0.0～1.0。当雾化因子的值为0时表示雾很浓，只看见雾，看不见物体。反之当雾化因子的值为1时，表示雾淡得已经看不见了，可以清晰地看到物体。
- dist为当前要绘制的片元离摄像机的距离。
- end表示一个特定的距离值，当片元距摄像机的距离超过end时，雾化因子为0。
- start也表示一个特定的距离值，当片元距摄像机的距离小于start时，雾化因子为1。

根据上述公式计算的雾化因子值与距离值之间的函数关系如图10-10所示。

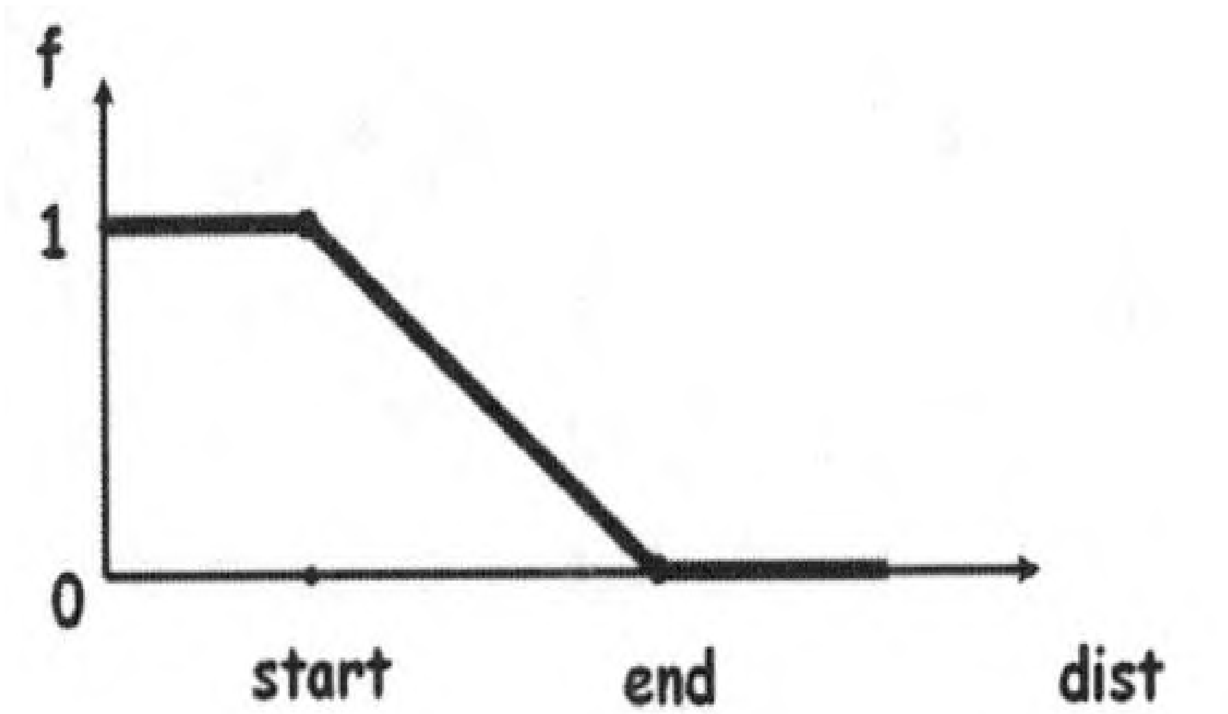


图10-10　雾化因子的线性变化

从图10-10中可以看出，雾化因子在start到end的范围内是线性变化的。但现实世界中的雾不完全是线性变化的，若希望模拟出更真实的雾，可以采用如下非线性的计算公式。

f=1.0-smoothstep（start,end,dist）

根据上述公式计算的雾化因子值与距离值之间的函数关系如图10-11所示，一般情况下采用此公式计算可以取得比线性公式更好的效果。

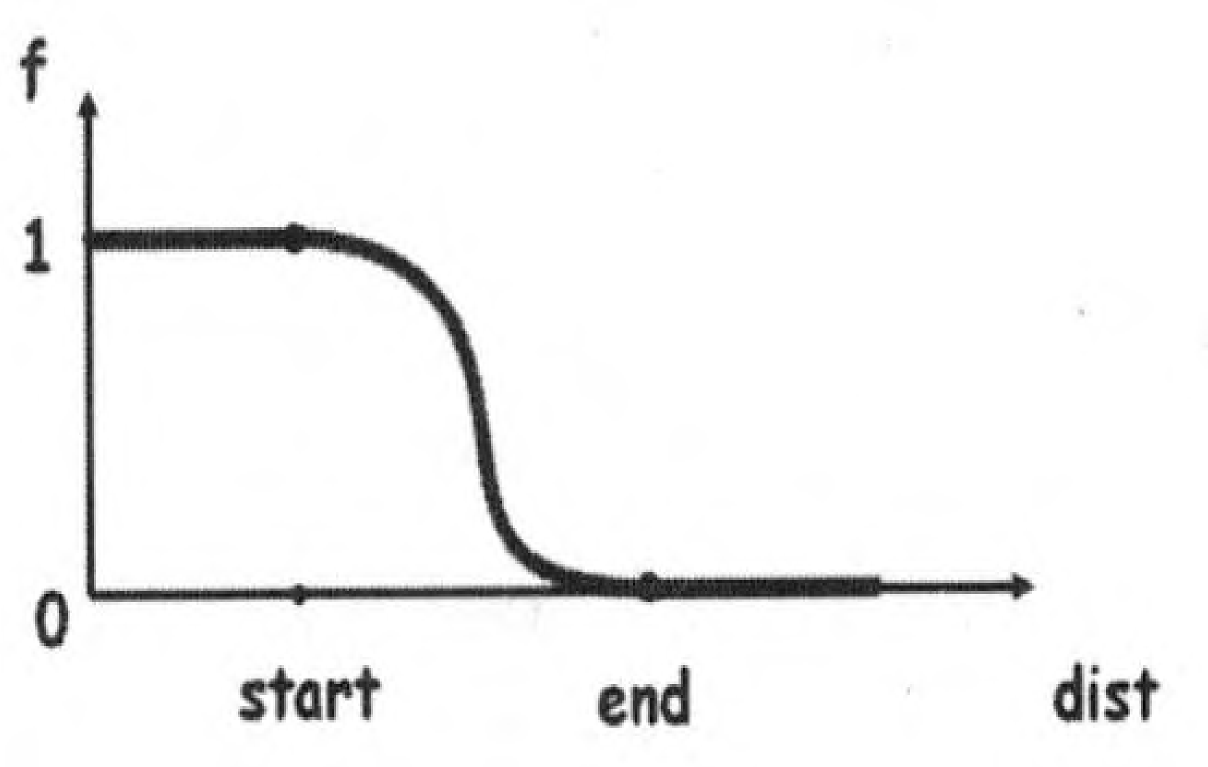


图10-11　雾化因子的非线性变化

10.3.2　雾的简单实现

了解了雾的原理与优势后，本小节将通过两个案例来向读者介绍如何在场景中应用雾。第一个案例（Sample10_3）采用的是线性计算模型，其运行效果如图10-12所示。

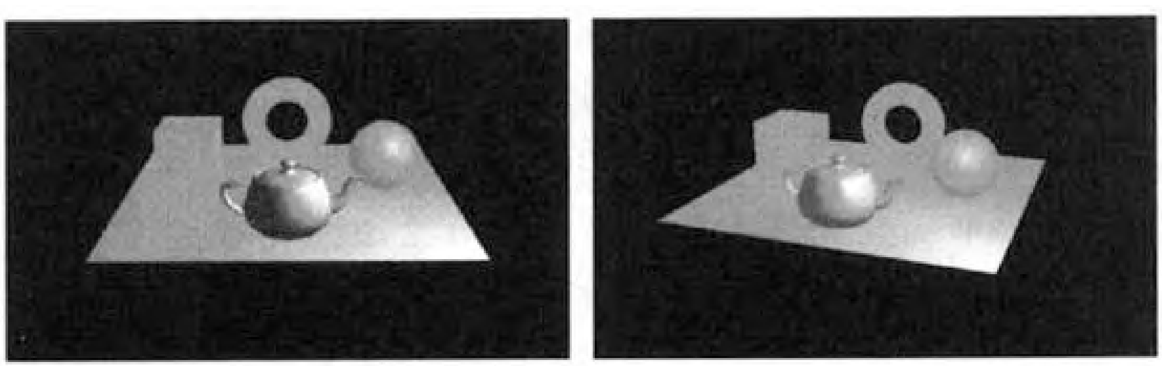


图10-12　案例Sample10_3的运行效果

提示

从图10-12中可以看出随着距离的增加物体逐渐融入了雾中。但由于本案例中的雾为黄色，经过灰度印刷后可能效果就不是很明显了，此时读者可以参考本书最前面的彩图或自行运行本案例进行观察。运行本案例时，若用手指在屏幕上左右滑动，可左右移动摄像机以变换视角观察。

了解了本案例的运行效果后，下面就可以进行具体的开发了。由于实际上本案例仅仅是将案例 Sample10_1复制了一份，去掉了滤光镜，并适当调整了场景中的物体和摄像机以及修改了着色器。因此这里仅介绍修改后着色器的代码，其他的部分请需要的读者参考随书光盘。

（1）首先介绍的是支持线性雾计算模型的顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第10章/Sample10_3/assets目录下的vertex_light.sh。

```
uniform mat4 uMVPMatrix;                          //总变换矩阵
uniform mat4 uMMatrix;                            //变换矩阵
uniform vec3 uLightLocation;                      //光源位置
uniform vec3 uCamera;                             //摄像机位置
attribute vec3 aPosition;                         //顶点位置
attribute vec3 aNormal;                           //顶点法向量
varying vec4 ambient;                             //用于传递给片元着色器的环境光最终强度
varying vec4 diffuse;                             //用于传递给片元着色器的散射光最终强度
varying vec4 specular;                            //用于传递给片元着色器的镜面光最终强度
varying float vFogFactor;                         //用于传递给片元着色器的雾化因子
……//此处省略了计算光照的pointLight方法，读者可自行查看随书光盘的源代码
float computeFogFactor(){                         //计算雾化因子的方法
  float tmpFactor;
  float fogDistance = length(uCamera-(uMMatrix*vec4(aPosition,1)).xyz);
                                                  //顶点到摄像机的距离
  const float end = 450.0;                        //雾结束位置
  const float start = 350.0;                      //雾开始位置
  tmpFactor = max(min((end- fogDistance)/(end-start),1.0),0.0);//用雾公式计算雾因子
  return tmpFactor;
}
void main() {
  gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点的位置
  pointLight(normalize(aNormal),ambient,diffuse,specular,    //计算光照
          uLightLocation,vec4(0.4,0.4,0.4,1.0),vec4(0.7,0.7,0.7,1.0),
          vec4(0.3,0.3,0.3,1.0));
  vFogFactor = computeFogFactor();          //将计算出的雾化因子值传递给片元着色器
}
```

说明

上述顶点着色器与原来案例中的大部分代码一致，主要是增加了12-19行的雾化因子计算方法computeFogFactor，此方法根据上一小节介绍的线性计算模型对应的公式计算雾化因子的值。

（2）介绍完顶点着色器后，下面介绍片元着色器，其代码如下。

代码位置：见随书光盘中源代码/第10章/Sample10_3/assets目录下的frag_light.sh。

```
precision mediump float;              //给出默认的浮点精度
varying vec4 ambient;                 //从顶点着色器传递过来的环境光最终强度
varying vec4 diffuse;                 //从顶点着色器传递过来的散射光最终强度
varying vec4 specular;                //从顶点着色器传递过来的镜面光最终强度
varying float vFogFactor;             //从顶点着色器传递过来的雾化因子
void main() {
vec4 objectColor=vec4(0.95,0.95,0.95,1.0);      //物体颜色
   vec4 fogColor = vec4(0.97,0.76,0.03,1.0);       //雾的颜色
   if(vFogFactor != 0.0){ //若物化因子不为0，则首先计算光照对物体颜色的影响，在与雾颜色加权
         objectColor = objectColor*ambient+objectColor*specular+objectColor*diffuse;
         gl_FragColor = objectColor*vFogFactor + fogColor*(1.0-vFogFactor);
      }else{       //如果雾因子为0，不必计算光照，不必加权，直接使用雾颜色作为片元颜色
         gl_FragColor=fogColor;
      }
   }}
```

提示

上述片元着色器收到雾化因子后，首先判断雾化因子是否为0，若为0则直接应用雾颜色作为最终片元颜色，不进行任何附加计算以提高性能。另外，由于本案例使用的是每顶点光照，因此采用了雾以后没有能将光照计算完全优化掉，若采用的是每片元光照则可以在雾因子为0时完全不进行光照计算，性能会优化得更多。

完成了采用线性雾计算模型的案例Sample10_3以后，只要将其复制一份并修改一下雾化因子的计算方法computeFogFactor即可得到采用非线性雾计算模型的案例Sample10_4，其运行效果如图10-13所示。

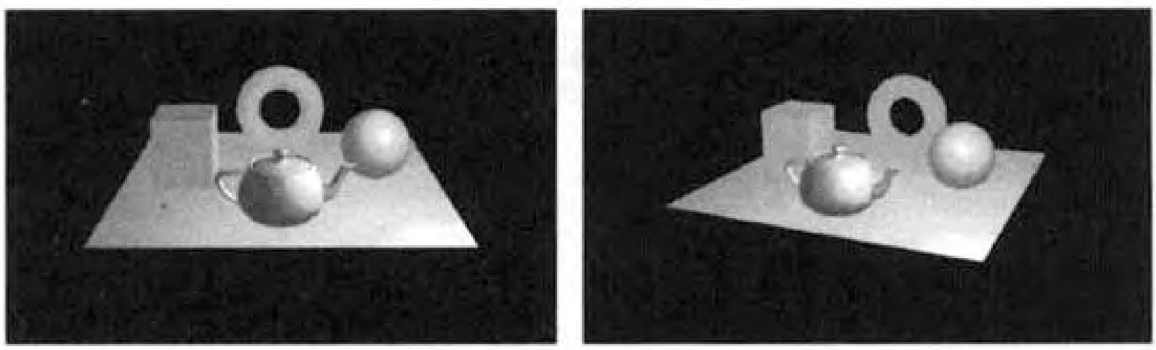


图10-13　案例Sample10_4的运行效果

提示

若细致比对两种不同的雾计算模型运行效果会发现，非线性的模型更加贴近现实世界，但区别不会很大。

了解了本案例的运行效果后，就可以进行具体的开发了。只要修改顶点着色器中的computeFogFactor方法即可，修改后computeFogFactor方法的代码如下。

代码位置：见随书光盘中源代码/第10章/Sample10_4/assets目录下的vertex_light.sh。

```
1 float computeFogFactor(){                                          //计算雾化因子的方法
2   float fogDistance = length(uCamera-(uMMatrix*vec4(aPosition,1)).xyz);//顶点到摄像机的距离
3   const float end = 490.0;                                         //雾结束位置
4   const float start = 350.0;                                       //雾开始位置
5   float tmpFactor = 1.0-smoothstep(start,end,fogDistance);         //计算雾化因子
6   return tmpFactor;
7 }
```

说明

上述computeFogFactor仅仅是将第5行的线性计算公式替换为了非线性计算公式，别的基本一致，没有太大变化。

本小节通过两个案例介绍了两种简单的雾计算模型，其实还有很多其他的、更复杂的雾计算模型，如$f=e^{-(density\times dist)}$ 或$f=e^{-(density\times dist)^2}$ 等。上述两个公式中的dist为待绘制片元到摄像机的距离，density为雾的浓度，有兴趣的读者也可以自行开发程序进行尝试。

10.4　本章小结

本章主要介绍了OpenGL ES 2.0中混合与雾的相关知识，通过本章的学习，读者应该可以根据需求开发出各种半透明效果，并可以使用雾效果为场景增加真实感。

第11章

常用3D开发技巧

前面的两章介绍了如何搭建各种形状的立体物体，掌握了这些知识读者已经可以搭建出不少较为真实的场景了。但要更真实地模拟现实世界还需要使用到很多其他的技术，本章将介绍一批在实际开发中非常常用的，主要包括：标志板、灰度图地形、天空盒、天空穹、镜像绘制、动态文本输出、非真实感绘制等。

11.1 标志板

模拟现实世界的场景中经常需要放置一些植物，如：树、灌木等。由于这些植物的外形是十分复杂的几何形状，若直接使用三角形

进行构建将需要海量的顶点，以现在主流智能手机或平板电脑的硬件配置是难以支撑的。因此构建场景中的植物时需要其他成本更为低廉的技术，本节介绍的标志板技术就是一种非常不错的选择。

11.1.1 案例效果与基本原理

标志板技术的基本原理非常简单，其使用纹理矩形来绘制植物，每棵植物仅需要一个纹理矩形即可描绘，其基本原理如图11-1所示。

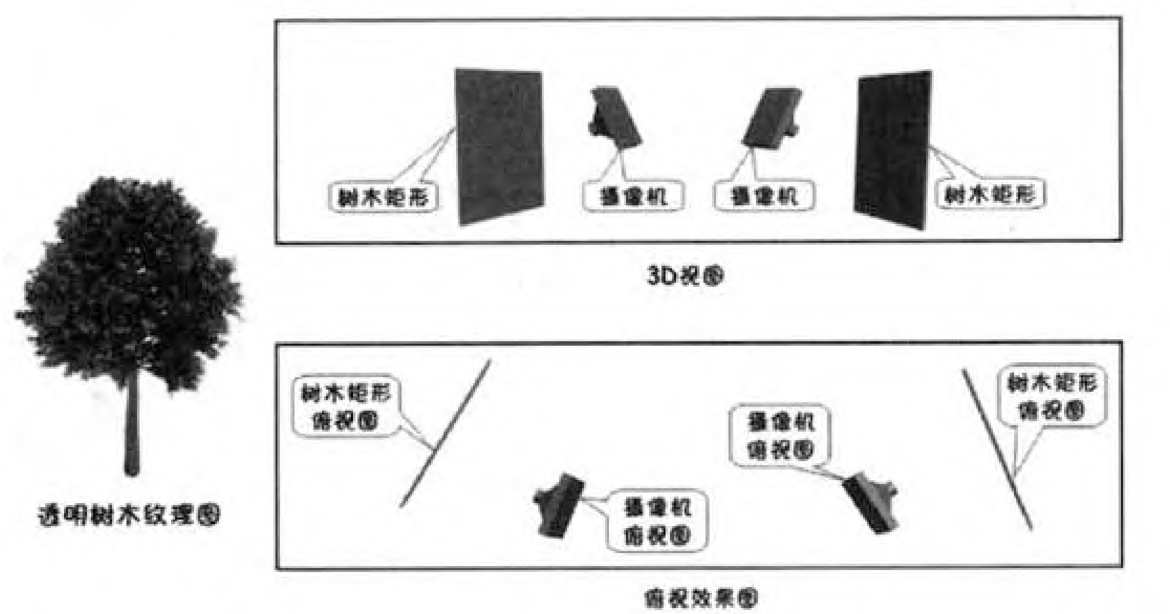


图11-1 标志版原理

从图11-1中可以看出，标志板技术的关键点如下。

- 每棵植物用一个纹理矩形进行绘制，纹理矩形上采用内容为植物的透明背景纹理图。绘制 纹理矩形时要采用恰当的混合因子，使植物产生正确的遮挡效果。
- 纹理矩形的朝向要根据当前摄像机的位置来动态决定，永远正对摄像机。

提示

由于基于标志板实现的植物实际是旋转的纹理矩形，因此适合用来呈现左右轴对称的植物，对于非左右轴对称的植物可能会给人

虚假的感觉。同时也正是由于每棵植物仅需要一个纹理矩形（若采用GL_TRIANGLES绘制方式仅需要6个顶点），所以此技术需要的系统资源非常少，效率很高。

由于标志板技术可以以非常低廉的成本呈现出较为真实的植物，因此在很多脍炙人口的游戏中都有采用，如图11-2和图11-3所示。

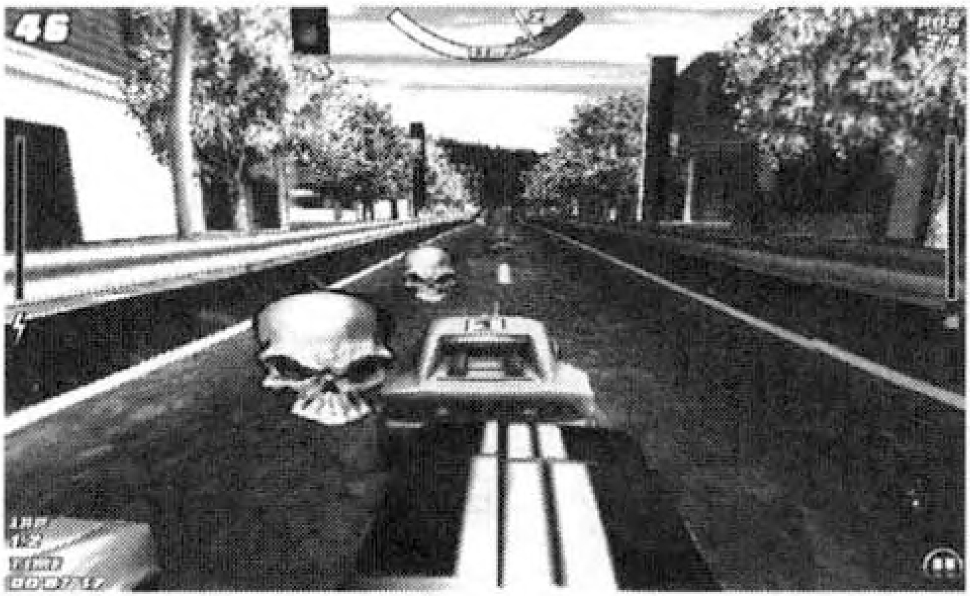


图11-2　Raging Thunder 2



图11-3　乡村飙车

了解了标志板技术的基本原理后，就可以进行本小节案例Sample11_1的开发了。开发前首先应该了解一下本案例的运行效果，如图11-4和图11-5所示。

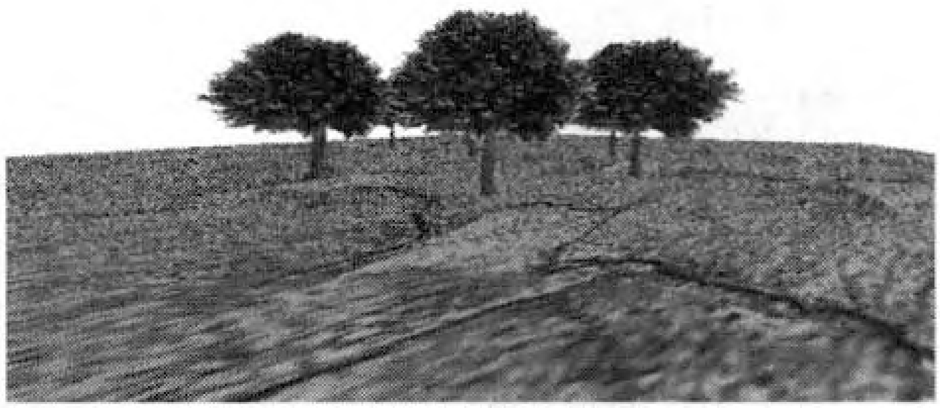


图11-4　远距离效果图

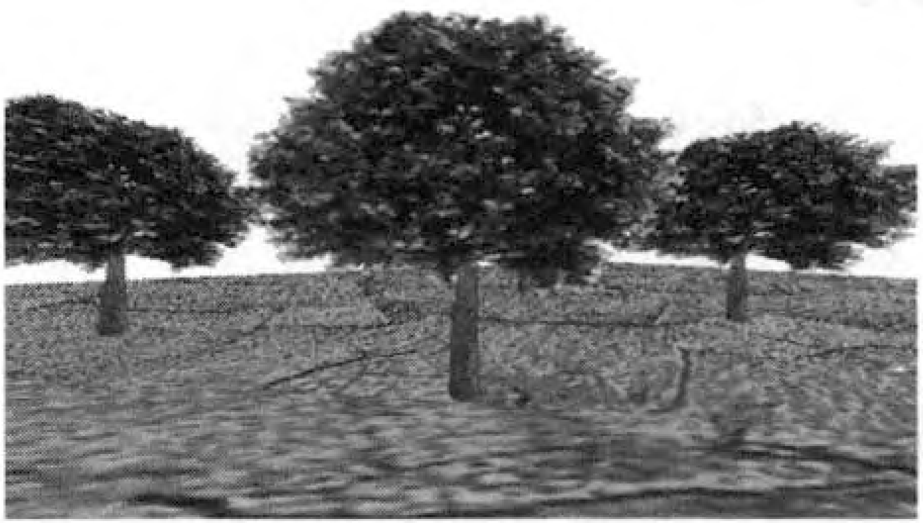


图11-5　近距离效果图

说明

图11-4为远距离效果图，而图11-5为近距离效果图。从两幅图的对比中可以看出，使用标志板技术呈现的植物在远处观察非常真实，但是当摄像机距离树木较近后，真实感有所下降，但尚能接受。另外，案例的场景中有多棵树，用手指按住屏幕左下角或右下

角，摄像机会绕场景旋转，用手指按住屏幕左上角或右上角摄像机会前进或后退，以从不同的位置、角度观察。

11.1.2 开发步骤

上一小节介绍了本案例的运行效果与基本原理，接下来在本小节中将介绍案例的具体开发步骤。由于本案例中使用到的部分类与前面章节很多案例中的基本一致，所以在本小节中不再对这些 相似的类进行重复介绍，仅给出本节案例中有代表性的几个类，具体内容如下。

（1）首先介绍表示单棵植物的SingleTree类，其实现了Comparable接口，具体代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_1/com/bn/Sample11_1目录下的SingleTree.java。

```java
package com.bn.Sample11_1;                                  //声明包名
import static com.bn.Sample11_1.MySurfaceView.*;            //导入相关类
public class SingleTree implements Comparable<SingleTree>{
    float x; float z;                                       //此植物的 x 与 z 坐标
    float yAngle;                                           //植物纹理矩形绕 y 轴旋转的角度
    TreeGroup tg;                                           //所属 TreeGroup 的引用
    public SingleTree(float x,float z,float yAngle,TreeGroup tg){
        this.x=x; this.z=z;                                 //初始化植物的 x 与 z 坐标
        this.yAngle=yAngle;                                 //初始化植物纹理矩形绕 y 轴旋转的角度
            this.tg=tg;                                     //记录所属 TreeGroup
        }
        public void drawSelf(int texId){                    //绘制单棵植物的 drawSelf 方法
            MatrixState.pushMatrix();                       //保护现场
            MatrixState.translate(x, 0, z);                 //平移到指定位置
            MatrixState.rotate(yAngle, 0, 1, 0);            //绕 y 轴旋转一定的角度
            tg.tfd.drawSelf(texId);         //调用 TreeForDraw 中的 drawSelf 方法绘制树木
            MatrixState.popMatrix();                        //恢复现场
        }
        public void calculateBillboardDirection(){//根据摄像机位置计算植物纹理矩形朝向的方法
            float xspan=x-cx;           //计算从植物位置到摄像机位置向量的 X 分量
            float zspan=z-cz;           //计算从植物位置到摄像机位置向量的 Z 分量
            if(zspan<=0){               //根据向量的两个分量计算出纹理矩形绕 Y 轴旋转的角度
                yAngle=(float)Math.toDegrees(Math.atan(xspan/zspan));
            }else{
                yAngle=180+(float)Math.toDegrees(Math.atan(xspan/zspan));
        } }
        @Override                       //根据植物到摄像机的距离比较两个植物"大小"的方法
        public int compareTo(SingleTree another){
            float xs=x-cx;              //计算从本植物位置到摄像机位置向量的 X 分量
            float zs=z-cz;              //计算从本植物位置到摄像机位置向量的 Z 分量
            float xo=another.x-cx;      //计算从另一植物位置到摄像机位置向量的 X 分量
            float zo=another.z-cz;      //计算从另一植物位置到摄像机位置向量的 Z 分量
            float disA=(float)Math.sqrt(xs*xs+zs*zs);    //计算当前植物到摄像机的距离
            float disB=(float)Math.sqrt(xo*xo+zo*zo);    //计算另一植物到摄像机的距离
            return ((disA-disB)==0)?0:((disA-disB)>0)?-1:1; //根据距离大小决定方法返回值
    } }
```

- 第4-7行为成员变量的声明，包括植物的x坐标、z坐标、绕y轴的旋转角度以及所属TreeGroup的引用。
- 第12-18行为用来绘制植物的drawSelf方法，此方法中首先将坐标系平移、旋转到指定的姿态，然后调用呈现植物的纹理矩形对象进行绘制。
- 第19-26行为计算植物纹理矩形朝向的calculateBillboardDirection方法，该方法根据植物的 位置及当前摄像机的位置计算出植物纹理矩形需要绕Y轴旋转的角度。
- 第28-36行为实现Comparable接口后重写的compareTo方法，此方法根据植物距摄像机的距离比较出植物的“大小”，供对一组植物进行排序时使用。对于Comparable接口不太熟悉的读者请参考其他专门介绍核心Java的书籍，这不是本书的侧重点，因此不予详述。

（2）接着介绍的是表示一组植物的TreeGroup类，其代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_1/com/bn/Sample11_1目录下的TreeGroup.java。

```
1 package com.bn.Sample11_1;                                  //声明包名
2 ……//此处省略了部分类的引入代码，读者可自行查看随书光盘中的源代码
3 public class TreeGroup{
4     TreeForDraw tfd;                                        //声明TreeForDraw类的引用
5     List<SingleTree> alist=new ArrayList<SingleTree>();  //创建SingleTree对象列表
6     public TreeGroup(MySurfaceView mv){
7         tfd=new TreeForDraw(mv);                            //创建TreeForDraw对象
8         alist.add(new SingleTree(0,0,0,this)); //创建的一个SingleTree对象并加入列表
9         ……//此处省略了部分创建SingleTree对象并添加进列表的代码
10        }
11        public void calculateBillboardDirection(){  //计算列表中每棵树的朝向
12            for(int i=0;i<alist.size();i++){        //循环遍历SingleTree对象列表
13                alist.get(i).calculateBillboardDirection();//计算每个植物纹理矩形的朝向
14       }}
15       public void drawSelf(int texId){              //绘制列表中的植物
16            for(int i=0;i<alist.size();i++){        //循环遍历SingleTree对象列表
17                alist.get(i).drawSelf(texId);//调用SingleTree对象的drawSelf方法绘制植物
18    }}}
```

- 第4-5行为本类成员变量的声明，主要包括TreeForDraw类的引用以及存储SingleTree对象的列表的引用。
- 第6-10行为本类的构造器，其中首先创建了TreeForDraw类的对象，接着向列表中添加了多个位于不同位置的SingleTree对象，每个SingleTree对象代表场景中的一株植物。
- 第11-14行为计算列表中每株植物对应纹理矩形朝向的calculateBillboardDirection方法，其通过遍历植物（SingleTree对象）列表，调用每个SingleTree对象的calculateBillboardDirection方法完成计算。
- 第15-18行为绘制植物列表中所有植物的drawSelf方法，其通过遍历植物（SingleTree对象）列表，调用每个SingleTree对象的

drawSelf方法完成绘制。

> **提示**
>
> TreeForDraw类其实就是一个普通的纹理矩形，与第7章很多案例中的基本一致，这里不再赘述，需要的读者请参考随书光盘中的源代码。

（3）最后需要介绍的是绘制整个场景的MySurfaceView类，其代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_1/com/bn/Sample11_1目录下的MySurfaceView.java。

```
package com.bn.Sample11_1;                                  //声明包名
```

```java
......//此处省略了部分类的导入代码，读者可以自行查阅随书光盘中的源代码
public class MySurfaceView extends GLSurfaceView{
    ......//此处省略了本类成员变量的声明，读者可以自行查阅随书光盘中的源代码
    ......//此处省略了本类的构造器，读者可以自行查阅随书光盘中的源代码
    @Override
    public boolean onTouchEvent(MotionEvent event){       //重写的触控事件处理方法
        ......//此处省略了本方法中的部分代码，读者可以自行查阅随书光盘中的源代码
        cx=(float)(Math.sin(direction)*Offset);           //计算新的摄像机 X 坐标
            cz=(float)(Math.cos(direction)*Offset);       //计算新的摄像机 Z 坐标
            mRender.tg.calculateBillboardDirection();  //重新计算所有植物纹理矩形的朝向
            Collections.sort(mRender.tg.alist);//将植物列表按照离摄像机的距离由远及近排序
            MatrixState.setCamera(cx,0,cz,0,0,0,0,1,0); //重新设定摄像机位置
            return true;
        }
        private class SceneRenderer implements GLSurfaceView.Renderer{
            TreeGroup tg;                                //声明植物集合 TreeGroup 的引用
            Desert desert;                               //沙漠类 Desert 的引用
            int treeId;                                  //树木纹理的 id
            int desertId;                                //沙漠纹理的 id
            @Override
            public void onDrawFrame(GL10 gl){          //绘制一帧画面的方法
             GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT|
                        GLES20.GL_COLOR_BUFFER_BIT);     //清除深度缓冲与颜色缓冲
             ......//此处省略了绘制沙漠的代码，读者可以自行查看随书光盘
             GLES20.glEnable(GLES20.GL_BLEND);              //开启混合
             GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA,
                  GLES20.GL_ONE_MINUS_SRC_ALPHA);           //设置混合因子
             MatrixState.pushMatrix();                      //保护现场
             MatrixState.translate(0, -2, 0);               //平移变换
             tg.drawSelf(treeId);                           //绘制所有植物
             MatrixState.popMatrix();                       //恢复现场
             GLES20.glDisable(GLES20.GL_BLEND);             //关闭混合
            }
            ......//此处省略了 onSurfaceChanged 方法，读者可以自行查阅随书光盘中的源代码
            @Override
            public void onSurfaceCreated(GL10 gl, EGLConfig config){
             GLES20.glClearColor(1.0f,1.0f,1.0f,1.0f);      //设置屏幕背景色 RGBA
             GLES20.glEnable(GLES20.GL_DEPTH_TEST);         //打开深度检测
             tg=new TreeGroup(MySurfaceView.this);          //创建 TreeGroup 的对象
             Collections.sort(mRender.tg.alist);            //初始化树顺序
             ......//此处省略了本方法中的部分代码，读者可以自行查阅随书光盘中的源代码
        } }
        ......//此处省略了加载纹理的方法，读者可以自行查阅随书光盘中的源代码
    }
```

- 第7-15行为重写的触控事件处理方法，该方法根据触控区域的不同对摄像机位置进行不同的变换。产生了摄像机的新位置后，根据列表中的植物距新摄像机位置的距离将列表中的植物由远及近进行排序。
- 第22-34行为绘制一帧画面的onDrawFrame方法，该方法首先绘制了表示沙漠的纹理矩形，接着在混合模式下绘制了列表中所有的植物。之所以要采用混合是因为植物是用纹理矩形呈现的，但植物对后面物体的遮挡不应该是整体的矩形，而应该在没有植物内容的位置透出后面的物体。

> **提示**
>
> 本案例中用到的着色器与第7章案例中的普通纹理映射着色器完全相同，这里不再赘述，需要的读者请参考随书光盘中的源代码。

上述案例中每次摄像机位置变化后都需要对列表中的植物按照离摄像机的远近重新排序，这是因为绘制植物时采用了混合。采用混合时想达到部分透明的效果必须先绘制被遮挡的物体，后绘制部分透明的遮挡面，不能再像绘制真正的立体物体那样仅依赖深度检测而不关心绘制顺序了。

若不将需要绘制的植物由远及近进行排序，以随意顺序绘制，在某些情况下就会产生不正确遮挡的视觉效果，如图11-6和图11-7所示。

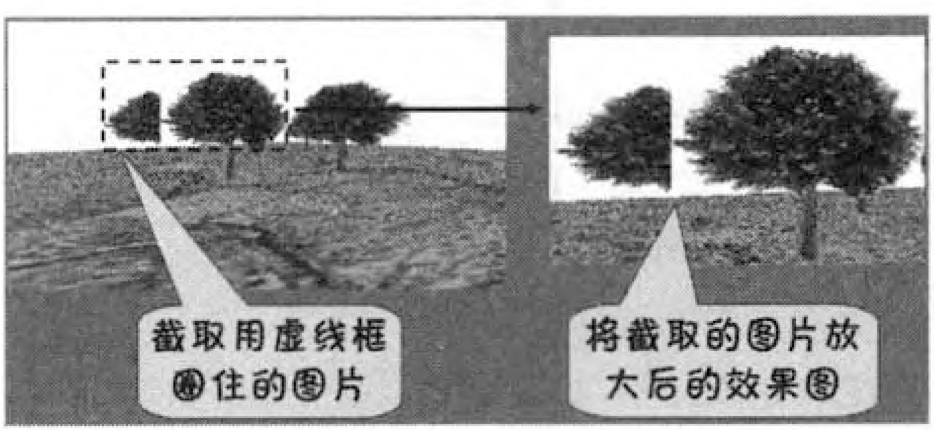


图11-6　未排序的错误效果1

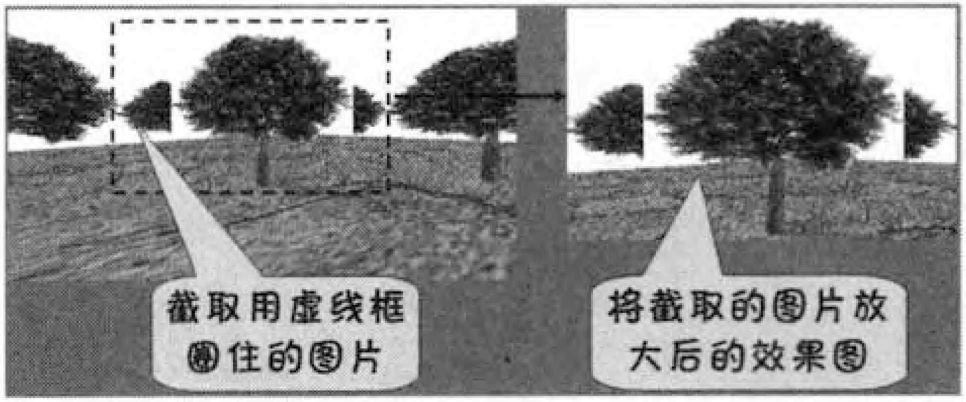


图11-7　未排序的错误效果2

产生图11-6与图11-7中不正确遮挡视觉效果的原因是，离摄像机近的植物对应的纹理矩形先被绘制了，此纹理矩形中的片元对应的深度缓冲已经记录了较小的深度值。再绘制距离远的植物对应的纹理矩形时，其中与距离近的片元重叠的片元深度检测就不会通过，将被丢弃。而不会与原有的距离近的片元进行混合，因此就不会产生距离较近的植物纹理矩形中透明的部分可以看到后面植物的情况。

正确的情况下，首先应该绘制远处的植物，此时在深度缓冲中记录的是较大的深度值。再绘制近处的植物时由于深度较浅，深度检测可以顺利通过，近处植物对应纹理矩形中的片元将与原有的片元执行混合。在正确设置了混合因子的情况下，透明的片元处就透出了后面的内容，产生了正确的遮挡效果。

提示

在视觉质量要求很高的场景中，当绘制植物时仅采用标志板就不能完全满足需要了，但全部植物都采用真实的3D模型进行绘制成本又太高。此时可以采用混合的绘制策略，当植物距离摄像机较近时采用真实的3D模型执行绘制，当距离摄像机较远时采用标志板执行绘制，这样既顾及了视觉效果又照顾到了执行成本。

11.2　灰度图地形

模拟现实世界的很多游戏场景中都需要用到地形，而自然界的地形形状非常复杂，直接由开发人员手工给出构成地形的每个三角形的顶点位置几乎是不可能的，甚至于采用3DMax这样的设计 工具也难以为继。因此3D开发领域的历代大牛发明了各种各样的地形生成技术，本节将要介绍的灰度图地形就是其中最为简便与常用的之一。

11.2.1　基本原理

灰度图地形生成技术的基本思想是用N×N的网格表示地形，同时提供一幅对应尺寸的灰度图。根据灰度图中每个像素的灰度来确定网格中顶点的海拔，黑色像素（RGB各个色彩通道的值为0）代表海拔最低的位置，白色像素（RGB各个色彩通道的值为255）代表海拔最高的位置，具体情况如图11-8所示。

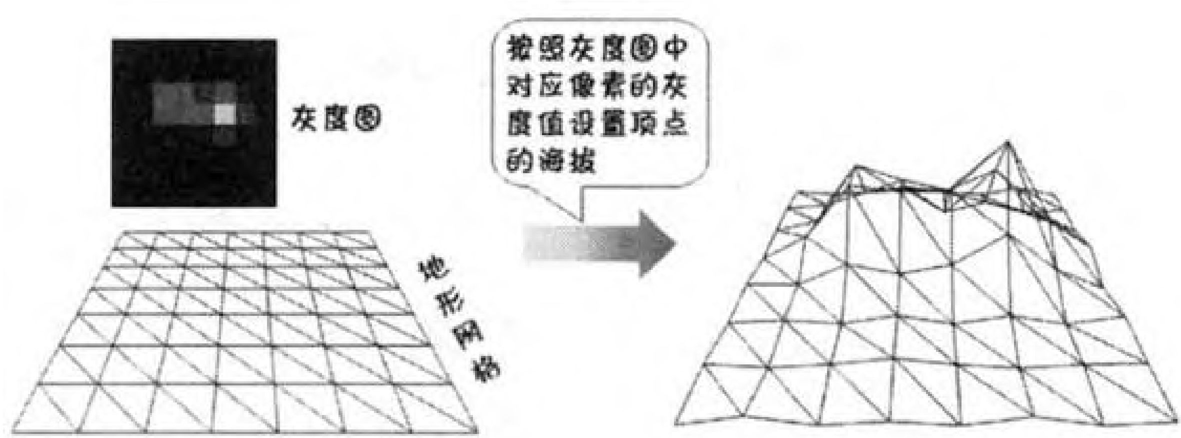


图11-8　灰度图地形技术原理示意图

具体开发中可以采用如下的公式来计算某像素顶点的海拔高度：

实际海拔=最低海拔+最大高差×像素值/255.0

> **提示**
>
> 要注意的是此公式中像素颜色的取值范围为0-255，而不是着色器中的0.0~1.0。

基于此技术生成地形时只需要采用绘图工具（如PhotoShop）用不同的灰度绘制出地形的海拔情况即可，非常简便与高效，图11-9所示就给出了本节案例中采用的一幅地形灰度图。

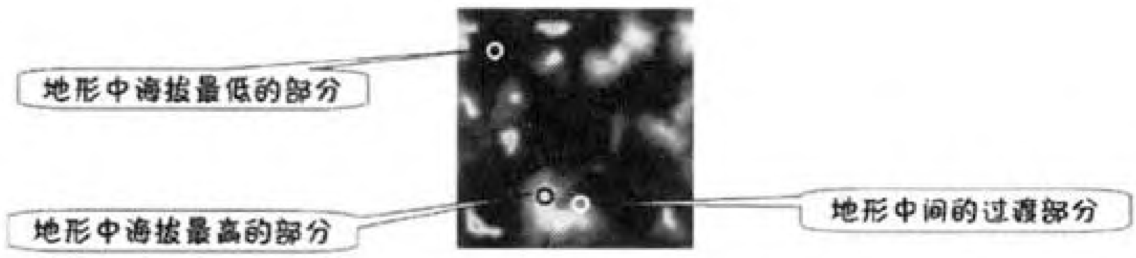
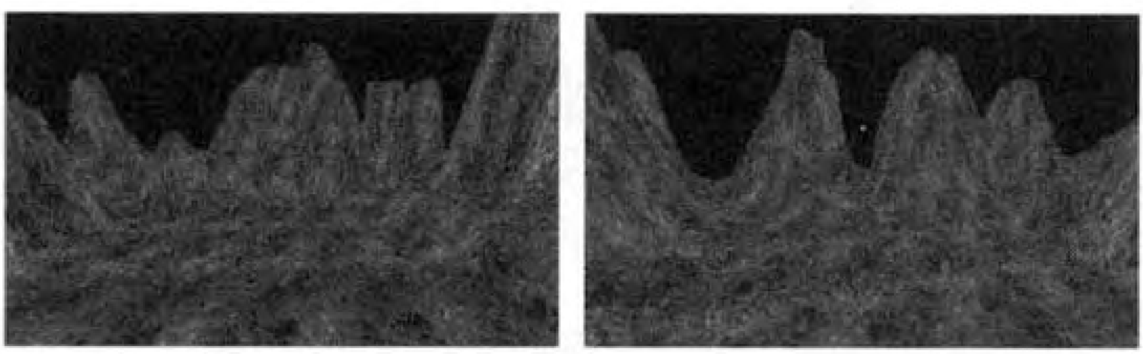


图11-9　本节案例中采用的一幅地形灰度图

11.2.2　普通灰度图地形

上一小节介绍了灰度图地形技术的基本原理，本小节将给出一个通过灰度图技术实现山地地形的案例Sample11_2，其运行效果如图11-10所示。


图11-10　案例Sample11_2的运行效果

> **提示**
>
> 用手指按住屏幕左下角或右下角，摄像机会绕场景旋转；用手指按住屏幕左上角或右上角，摄像机会前进或后退，以从不同的位置、角度观察场景。

由于本案例中使用的部分类与前面章节很多案例中的基本一致，所以在本小节中不再对这些相似的类进行重复地介绍，仅给出本节案

例中有代表性的几个类，具体内容如下。

（1）为了代码开发与维护的方便，本小节的案例将其中很多类都需要用到的常量集中到了常量 类Constant中，其代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_2/com/bn/Sample11_2目录下的Constant.java。

```
1package com.bn.Sample11_2;                              //声明包名
2……//此处省略了部分包的导入代码，读者可以自行查阅随书光盘中的源代码
3public class Constant{
4     public static float[][] yArray;                    //存储地形顶点高度的数组
5     public static final float LAND_HIGH_ADJUST=-2f; //陆地的最低海拔
6     public static final float LAND_HIGHEST=20f;       //陆地最大高差
7     public static float[][] loadLandforms(Resources resources,int index){
8         Bitmap bt=BitmapFactory.decodeResource(resources, index);     //导入灰度图
9         int colsPlusOne=bt.getWidth();                  //获得存储地形顶点高度数组的列数
10            int rowsPlusOne=bt.getHeight();             //获得存储地形顶点高度数组的行数
11            float[][] result=new float[rowsPlusOne][colsPlusOne];//创建存储地形顶点高度的数组
12            for(int i=0;i<rowsPlusOne;i++){             //对灰度图像素行遍历
13             for(int j=0;j<colsPlusOne;j++){            //对灰度图像素列遍历
14               int color=bt.getPixel(j,i);              //获得指定行列处像素的颜色
15               int r=Color.red(color);                  //获取该像素红色通道的值
16               int g=Color.green(color);                //获取该像素绿色通道的值
17               int b=Color.blue(color);                 //获取该像素蓝色通道的值
18               int h=(r+g+b)/3;                         //3个色彩通道求平均
19               result[i][j]=h*LAND_HIGHEST/255+LAND_HIGH_ADJUST;//按公式计算顶点海拔
20            }}
21            return result;                              //返回存储地形顶点高度的数组
22    }}
```

- 第4-6行为本类的成员变量，主要包括存储地形顶点高度的数组、陆地的最低海拔以及陆地最大高差。
- 第7-27行为加载灰度图并根据灰度图中各个像素的灰度值计算对应顶点海拔高度的loadLandforms方法，其最终将计算出的顶点海拔以二维数组的形式返回。

（2）介绍完常量类后，下面介绍的是用来绘制山地地形的Mountion类，其代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_2/com/bn/Sample11_2目录下的Mountion.java。

```
package com.bn.Sample11_2;                          //声明包名
......//此处省略了部分类的导入代码，读者可以自行查阅随书光盘中的源代码
public class Mountion{
    float UNIT_SIZE=1.0f;                           //地形网格中每个小格子的尺寸
    ......//此处省略了本类的部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
    int vCount=0;                                   //地形中顶点的数量
    ......//此处省略了本类的构造器，读者可以自行查阅随书光盘中的源代码
   public void initVertexData(float[][] yArray,int rows,int cols){//初始化顶点数据的方法
        vCount=cols*rows*2*3;           //每个小格子两个三角形，每个三角形3个顶点
            float vertices[]=new float[vCount*3];   //存储顶点X、Y、Z坐标的数组
            int count=0;                            //顶点计数器
            for(int j=0;j<rows;j++){                //遍历地形网格的行
            for(int i=0;i<cols;i++){                    //遍历地形网格的列
             float zsx=-UNIT_SIZE*cols/2+i*UNIT_SIZE;   //计算当前小格子左上侧点的x坐标
```

```
            float zsz=-UNIT_SIZE*rows/2+j*UNIT_SIZE;    //计算当前小格子左上侧点的 z 坐标
            //将当前行列对应的小格子中顶点的坐标按照卷绕成两个三角形的顺序存入顶点坐标数组
           vertices[count++]=zsx; vertices[count++]=yArray[j][i]; vertices[count++]
           =zsz;
            ......//此处省略了一些将顶点坐标存入数组的代码，与上面的类似
           } }
           ......//此处省略了创建顶点数据缓冲区的代码，读者可以自行查阅随书光盘中的源代码
           float[] texCoor=generateTexCoor(cols,rows); //生成顶点纹理坐标数组
           ByteBuffer cbb = ByteBuffer.allocateDirect(texCoor.length*4);
                                                         //创建纹理坐标数据缓冲
           cbb.order(ByteOrder.nativeOrder());          //设置字节顺序
           mTexCoorBuffer = cbb.asFloatBuffer();        //转换为 Float 型缓冲
           mTexCoorBuffer.put(texCoor);                 //向缓冲区中存入纹理坐标
           mTexCoorBuffer.position(0);                  //设置缓冲区起始位置
     }
     ......//此处省略了初始化着色器的方法，读者可以自行查阅随书光盘中的源代码
     ......//此处省略了绘制山地地形的 drawSelf 方法，读者可以自行查阅随书光盘中的源代码
    public float[] generateTexCoor(int bw,int bh){//根据网格行列数自动产生纹理坐标数组的方法
           float[] result=new float[bw*bh*6*2];   //创建存放纹理坐标值的数组
           float sizew=8.0f/bw;                   //列数，8.0 代表 S 轴纹理坐标最大值
           float sizeh=8.0f/bh;                   //行数，8.0 代表 T 轴纹理坐标最大值
           int c=0;                               //坐标计数器
           for(int i=0;i<bh;i++){                 //对行遍历
            for(int j=0;j<bw;j++){                //对列遍历
               float s=j*sizew; float t=i*sizeh;
                                    //每个矩形由两个三角形构成，共 6 个点，12 个纹理坐标
               result[c++]=s; result[c++]=t;                 //左上角顶点纹理坐标
               result[c++]=s; result[c++]=t+sizeh;           //左下角顶点纹理坐标
               result[c++]=s+sizew; result[c++]=t;           //右上角顶点纹理坐标
               result[c++]=s+sizew; result[c++]=t;           //右上角顶点纹理坐标
               result[c++]=s; result[c++]=t+sizeh;           //左下角顶点纹理坐标
               result[c++]=s+sizew; result[c++]=t+sizeh;     //右下角顶点纹理坐标
           } }
           return result;                                    //将结果纹理坐标返回
} }
```

- 第9-19行为计算地形网格中各个顶点坐标，并根据卷绕成三角形的需要将顶点坐标存入顶点坐标数组的代码。顶点的x、z坐标值是根据顶点所处的行、列以及每个格子的长度值计算得出的，而顶点的y坐标是根据顶点所处的行、列从顶点高度数组中获取的。顶点高度数组是通过Constant类的loadLandforms方法从灰度图得出的。
- 第30-46行为根据地形网格行列数自动产生纹理坐标数组的generateTexCoor方法，该方法根据地形网格的行、列以及最大纹理值计算出每个顶点对应的纹理坐标。

提示

本案例中纹理坐标S轴与T轴的最大值都是8.0，这意味着纹理将在整个山体中被重复8次。当然想达到重复8次的目的还需要在加载纹理时将纹理的拉伸方式设置为GL_REPEAT。

（3）介绍完用来绘制山地地形的Mountion类后，接着介绍的是用于绘制场景的MySurfaceView类，其代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_2/com/bn/Sample11_2目录下的MySurfaceView.java。

```
package com.bn.Sample11_2;                                     //声明包名
......//此处省略了部分类的导入代码，读者可以自行查阅随书光盘中的源代码
public class MySurfaceView extends GLSurfaceView{
    ......//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
    ......//此处省略了本类的构造器以及触控监听方法，读者可以自行查阅随书光盘中的源代码
  private class SceneRenderer implements GLSurfaceView.Renderer{
    Mountion mountion;                                         //山地地形对象引用
    int mountionId;                                            //山地草皮纹理id
    ......//此处省略了onDrawFrame与onSurfaceChanged方法，读者可以查阅光盘中的源代码
        @Override
        public void onSurfaceCreated(GL10 gl, EGLConfig config){
          ......//此处省略了本方法中的部分代码，读者可以自行查阅随书光盘中的源代码
          //创建山地地形对象，其构造器入口参数主要为从灰度图中加载的海拔高度数据数组相关
          mountion=new Mountion(MySurfaceView.this,yArray,yArray.length-1,yArray[0].
         length-1);
          mountionId=initTexture(R.drawable.grass);      //加载山地地形草皮纹理
        }}
        ......//此处省略了加载纹理的initTexture方法，读者可以自行查阅随书光盘中的源代码
  }
```

- 上述MySurfaceView类与本书前面很多案例中已经出现的MySurfaceView类非常类似，主要工作都是创建场景中的物体、加载纹理、设置相关绘制参数等。
- 区别较大的一处就是创建山地地形对象（Mountion类对象）时，其中顶点的Y坐标来自从灰度图加载生成的海拔高度数组。

提示

本案例中用到的着色器与第7章案例中的普通纹理映射着色器完全相同，这里不再赘述，需要的读者请参考随书光盘中的源代码。

11.2.3　过程纹理地形

上一小节的案例中通过灰度图地形生成技术渲染呈现了一个山地地型的场景，从运行效果中可以看出，呈现的地形还是比较真实的。但有一个明显的不足，场景中的山体从上到下都是一种外观，不符合现实世界的情况。

现实世界中的山体一般有海拔不同、外观不同的规律，因此本小节将对上一小节的案例进行升级，对山体采用过程纹理技术进行渲染呈现。升级后的案例为Sample11_3，其运行效果如图11-11所示。

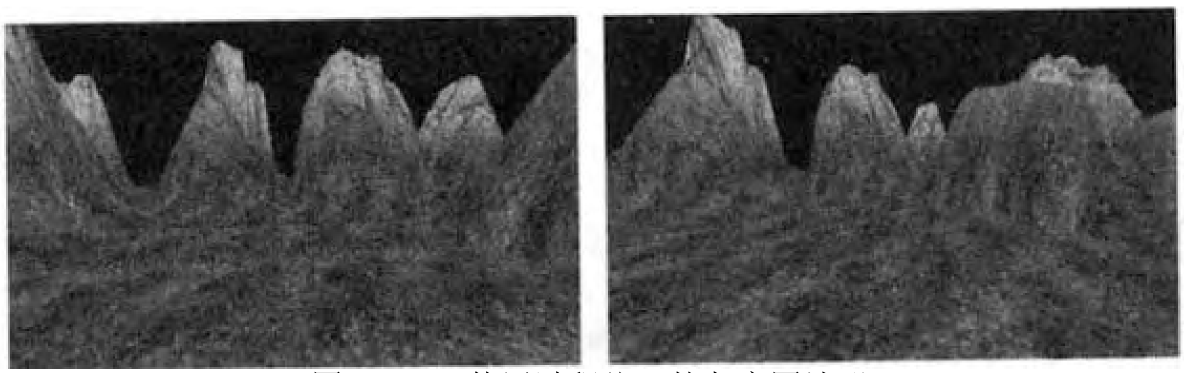


图11-11　使用过程纹理的灰度图地形

提示

由于本书中的插图采用灰度印刷，可能看不出过程纹理的效果，此时请读者参照本书最前面的彩页或自行运行程序观察。

了解了本案例的运行效果后，下面就可以进行代码的开发了。由于实际上本案例仅仅是将前面 小节的案例Sample11_2复制了一份并进行了修改，因此这里仅给出主要的修改步骤，具体如下所列。

（1）首先需要修改的是山地地形类Mountion，其代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_3/com/bn/Sample11_3目录下的Mountion.java。

```
package com.bn.Sample11_3;                              //声明包名
......//此处省略了部分类的导入代码，读者可以自行查阅随书光盘中的源代码
public class Mountion{
    ......//此处省略了本类的部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
    int sTextureGrassHandle;                            //草皮纹理的引用
    int sTextureRockHandle;                             //岩石纹理的引用
    int landStartYYHandle;                              //过程纹理起始 Y 坐标的引用
    int landYSpanHandle;                                //过程纹理跨度的引用
    ......//此处省略了本类的构造器、初始化顶点数据的 initVertexData 方法、
        ......//需要的读者可以查阅随书光盘中的源代码
        public void initShader(MySurfaceView mv){   //初始化着色器的方法
            ......//此处省略了部分与升级前一致的代码，读者可以自行查阅随书光盘中的源代码
            sTextureGrassHandle=GLES20.glGetUniformLocation(
             mProgram, "sTextureGrass");            //获取草皮纹理引用值
            sTextureRockHandle=GLES20.glGetUniformLocation(
             mProgram, "sTextureRock");             //获取岩石纹理引用值
            landStartYYHandle=GLES20.glGetUniformLocation(
             mProgram, "landStartY");               //获取过程纹理起始 Y 坐标引用值
            landYSpanHandle=GLES20.glGetUniformLocation(
             mProgram, "landYSpan");                //获取过程纹理跨度的引用值
        }
        public void drawSelf(int texId,int rock_textId){//绘制山地地形的方法
            ......//此处省略了部分与升级前一致的代码，读者可以自行查阅随书光盘中的源代码
            GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
            GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, texId);  //绑定草皮纹理
            GLES20.glActiveTexture(GLES20.GL_TEXTURE1);
            GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, rock_textId);//绑定岩石纹理
            GLES20.glUniform1i(sTextureGrassHandle, 0); //草皮纹理编号为 0
            GLES20.glUniform1i(sTextureRockHandle, 1);  //岩石纹理编号为 1
            GLES20.glUniform1f(landStartYYHandle, 0);   //传送过程纹理起始 Y 坐标
            GLES20.glUniform1f(landYSpanHandle, 30);    //传送过程纹理跨度
            GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount);//绘制地形
        }
        ......//此处省略了根据网格行列数自动产生纹理坐标数组的 generateTexCoor 方法
}
```

- 第5-8行为新增加的4个成员变量声明，都是用于向渲染管线传递数据的引用，包括：草皮纹理的引用、岩石纹理的引用、过程纹理起始*Y* 坐标的引用、过程纹理跨度的引用。
- 第11-21行为初始化着色器的initShader方法，其中增加了获取用于向着色器传递数据的4个引用值的代码。
- 第22-33行为绘制山地地形的drawSelf方法，其中增加了绑定岩石纹理、为纹理分配编号、传送过程纹理起始*Y* 坐标进渲染管线、传送过程纹理跨度进渲染管线的相关代码。

提示

还需要在MySurfaceView类中增加加载岩石纹理、绘制时传递岩石纹理id给Mountion类drawSelf方法的相关代码。这些代码非常简单，有需要的读者请参考随书光盘。

（2）完成了Java代码的修改后，下面就需要修改着色器了。首先是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第11

章/Sample11_3/asset目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;           //总变换矩阵
attribute vec3 aPosition;          //顶点位置
attribute vec2 aTexCoor;           //顶点纹理坐标
varying vec2 vTextureCoord;        //用于传递给片元着色器的纹理坐标
varying float currY;               //用于传递给片元着色器的Y坐标
void main(){
  gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点的位置
  vTextureCoord = aTexCoor;                     //将接收的纹理坐标传递给片元着色器
     currY=aPosition.y;                         //将顶点的Y坐标传递给片元着色器
  }
```

说明

上述顶点着色器中主要是增加了将顶点Y坐标通过易变变量传递给片元着色器的代码。

（3）接着需要修改的是片元着色器，其代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_3/asset目录下的frag.sh。

```
precision mediump float;                          //给出默认的浮点精度
varying vec2 vTextureCoord;                       //接收从顶点着色器过来的纹理坐标
varying float currY;                              //接收从顶点着色器过来的 Y 坐标
uniform sampler2D sTextureGrass;                  //纹理内容数据（草皮）
uniform sampler2D sTextureRock;                   //纹理内容数据（岩石）
uniform float landStartY;                         //过程纹理起始 Y 坐标
uniform float landYSpan;                          //过程纹理跨度
void main(){
  vec4 gColor=texture2D(sTextureGrass, vTextureCoord);     //从草皮纹理中采样出颜色
     vec4 rColor=texture2D(sTextureRock, vTextureCoord);   //从岩石纹理中采样出颜色
     vec4 finalColor;                                      //最终颜色
     if(currY<landStartY){
        finalColor=gColor;  //当片元 Y 坐标小于过程纹理起始 Y 坐标时采用草皮纹理
```

```
     }else if(currY>landStartY+landYSpan){
        finalColor=rColor;  //当片元 Y 坐标大于过程纹理起始 Y 坐标加跨度时采用岩石纹理
     }else{
        float currYRatio=(currY-landStartY)/landYSpan; //计算岩石纹理所占的百分比
        finalColor= currYRatio*rColor+(1.0- currYRatio)*gColor;
                                                  //将岩石、草皮纹理颜色按比例混合
     }
         gl_FragColor = finalColor;                //给此片元最终颜色值
  }
```

- 上述片元着色器中不再是仅采用草皮纹理对山体进行着色，而是同时采用了两幅纹理。
- 当片元Y坐标小于过程纹理起始Y坐标时用草皮纹理着色，大于过程纹理起始Y坐标加跨度时采用岩石纹理着色。位于二者之间时

将草皮纹理、岩石纹理按照计算的比例进行混合，混合的规则为片元Y坐标越大，岩石纹理所占百分比越大。

> **提示**
>
> 本小节案例中对同一物体根据情况的不同应用不同的纹理进行着色，这种技术就是过程纹理技术。过程纹理技术更多的是一种解决问题的策略，不是一成不变的，实际开发中读者可以设计更多、更好的过程纹理计算模型。

11.2.4　MipMap地形

经过上一小节的升级后，山地地形场景的真实感增加了不少，但还有一个明显的瑕疵。那就是远处的地形会比近处的更清楚，这显然不符合我们观察现实世界的经验。造成这种现象的原因是案例中采用的纹理图仅有一套，无论山体远近都采用同一套纹理图进行纹理映射。

- 远处同样面积的山体投影到屏幕上所占的面积小，近处的投影到屏幕上所占的面积大。近处的山体进行纹理采样时会采用MAG方式（纹理图会被拉大），远处的山体进行纹理采样时会采用MIN方式（纹理图会被缩小）。
- 同样的纹理图拉大后就显得清晰度差些，缩小后就显得很锐利。

想改善这种不真实感非常简单，只要采用第7章介绍过的MipMap纹理技术就可以了。本小节将进一步对上一小节的案例进行升级，升级后的案例Sample11_4运行效果如图11-12所示。

加清晰的问题得到了很大程度的改善。若由于印刷的问题比对效果不明显，读者可以参考本书前面的彩图或自行运行案例进行观察。

提示

采用MipMap纹理不但可以改善地形中远处山体比近处山体更加清晰的问题，还可以提升执行性能，但需要的纹理空间接近普通纹理的2倍。

由于本案例仅仅是对前一小节案例的升级，因此很多同样的代码不再赘述，这里仅给出需要修 改的部分。那就是MySurfaceView类中加载纹理的initTexture方法，其代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_4/com/bn/Sample11_4目录下的MySurfaceView.java。

```
1 public int initTexture(int drawableId){                              //加载纹理的方法
2   ……//此处省略了与修改前相同的部分代码，需要的读者请参看光盘
3   GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureId); //绑定纹理
4   GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,GLES20.GL_TEXTURE_MAG_FILTER,
5               GLES20.GL_LINEAR_MIPMAP_LINEAR);   //使用MipMap线性纹理采样
6   GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,GLES20.GL_TEXTURE_MIN_FILTER,
7               GLES20.GL_LINEAR_MIPMAP_NEAREST);  //使用MipMap最近点纹理采样
8   ……//此处省略了与修改前相同的部分代码，需要的读者请参看光盘
9   GLES20.glGenerateMipmap(GLES20.GL_TEXTURE_2D); //自动生成Mipmap系列纹理
10      ……//此处省略了与修改前相同的部分代码，需要的读者请参看光盘
11  }
```

- 第3-7行是纹理采样方式的设置，将原来的普通纹理采样方式修改成了MipMap纹理采样方式。要注意的是MipMap纹理采样本身也分线性采样与最近点采样，读者可以在实际开发中根据需要选用。
- 第9行是新增的自动生成MipMap系列纹理图的代码。

> **提示**
>
> 请读者特别注意，部分机型要求MipMap纹理图必须长宽相等，否则不能正确显示。因此，笔者强烈建议读者采用长宽相等的纹理图用于MipMap纹理映射。

11.3 天空盒与天空穹

上一节通过灰度图地形、过程纹理、MipMap纹理等技术构建了非常真实的地形场景，但场景中的天空黑茫茫一片，成为一个明显的瑕疵。本节将向读者介绍两种用于实现天空效果的技术，天空盒与天空穹。采用这两种技术可以为场景添加真实的天空效果，大大增加场景的真实感。

11.3.1 天空盒

天空盒技术的思路非常简单，具体说就是将场景放置在一个很大的立方体中，立方体的每个面是一个纹理正方形，如图11-13所示。

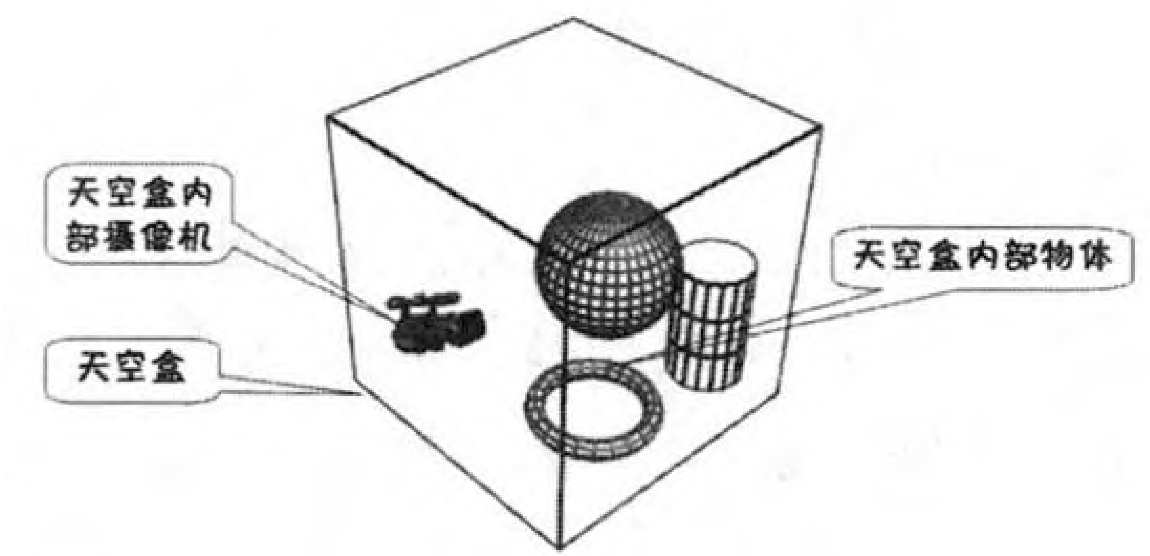


图11-13　天空盒示意图

> **提示**
>
> 使用天空盒时需要注意，用于观察场景的摄像机需要放置在天空盒立方体的内部，图11-13中就是如此。

为了在观察位于天空盒内部的场景时有真实天空背景的效果，组成天空盒的6个纹理正方形上需要各自映射一幅正方形的天空纹理图。这6幅纹理图是可以无缝拼接的，如图11-14所示。

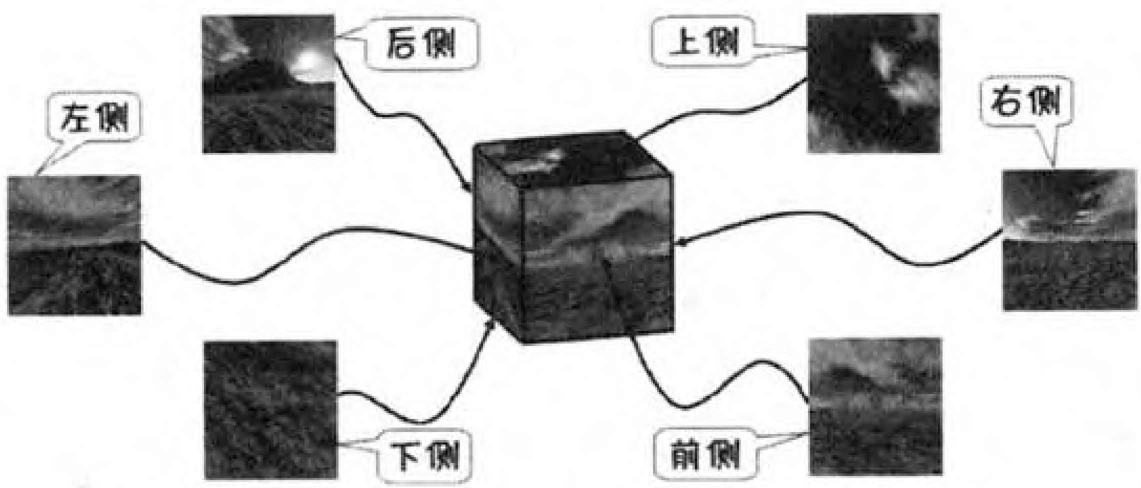


图11-14　天空盒纹理贴图示意图

了解了天空盒的原理后，就可以进行本小节案例Sample11_5的开发了。在介绍开发步骤之前 请读者首先了解一下本案例的运行效果，如图11-15所示。



图11-15　天空盒案例效果图

说明

图11-15中左右两幅分别为摄像机在天空盒中向不同的方向观察时看到的天空情况。运行本案例时当手指在屏幕上左右滑动时摄像机绕场景转动，当手指在屏幕上上下滑动时摄像机上升或下降。

通过前面的介绍读者已经知道，天空盒实际上是由6个纹理正方形（正方形为矩形的特殊情况）组成的。而在本书第7章介绍纹理的相关知识时已经对纹理矩形的开发进行了详细的介绍，因此这里仅给出将6个纹理矩形组装成天空盒的MySurfaceView类的相关代码。

代码位置：见随书光盘中源代码/第11章/Sample11_5/com/bn/Sample11_5目录下的MySurfaceView.java。

```
package com.bn.Sample11_5;                              //声明包名
.....//此处省略了部分类的导入代码，读者可以自行查阅随书光盘中的源代码
class MySurfaceView extends GLSurfaceView{
  .....//此处省略了本类部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
  int[] textureIdA=new int[6];                          //存储天空盒 6 面的纹理 id 的数组
   .....//此处省略了本类的构造器，读者可以自行那个查阅随书光盘中的源代码
   .....//此处省略了触控事件的处理方法 onTouchEvent，读者可以查阅随书光盘中的源代码
   private class SceneRenderer implements GLSurfaceView.Renderer{
          TextureRect texRect;                          //用于绘制天空盒各个面的纹理正方形
          public void onDrawFrame(GL10 gl){             //绘制一帧画面的方法
           GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT|
             GLES20.GL_COLOR_BUFFER_BIT);               //清除深度缓冲与颜色缓冲
           MatrixState.setCamera(cx,cy,cz,0f,0f,0f,0f,1.0f,0.0f); //设置摄像机
           final float tzz=0.4f;                                  //天空盒 6 个面的调整值
           MatrixState.pushMatrix();                              //保护现场
           MatrixState.translate(0, 0, -UNIT_SIZE+tzz);           //平移变换
           texRect.drawSelf(textureIdA[0]);                       //绘制天空盒的后侧面
           MatrixState.popMatrix();                               //恢复现场
         .....//此处省略了绘制天空盒前侧、左侧、右侧以及下侧的代码，需要时请参考光盘
           MatrixState.pushMatrix();                              //保护现场
           MatrixState.translate(0, UNIT_SIZE-tzz, 0);            //平移变换
           MatrixState.rotate(90, 1, 0, 0);                       //旋转变换
           texRect.drawSelf(textureIdA[4]);                       //绘制天空盒上侧面
           MatrixState.popMatrix();                               //恢复现场
         }
```

```
            ....//此处省略了重写的onSurfaceChangedd方法，需要时请参考光盘
            ....//此处省略了重写的onSurfaceCreated方法，需要时请参考光盘
        }
        ....//此处省略了自动生成纹理id的方法，读者可以自行查阅随书光盘中的源代码
}
```

说明

从上述代码中可以看出，通过6个纹理正方形组装天空盒是非常简单的，只要将一个纹理正方形对象通过平移、旋转等变换到指定的位置，再应用不同的纹理进行绘制即可。另外在实际的开发中，建议读者将天空盒的相关代码封装到一个单独的类中，而不应该像本案例一样直接将相关代码写在绘制一帧画面的onDrawFrame方法中，这样不便于管理和重用代码。

11.3.2　天空穹

细心的读者可能会发现，从大部分角度观察天空盒都是比较真实的，但当观察天空盒任何两个面的接缝处时真实感就差很多。这是因为构成接缝的两个面成90度角，不是平滑的。这个问题是天空盒技术所固有的，很难彻底解决，因此很多游戏场景中会采用另一种天空效果实现技术一天空穹。

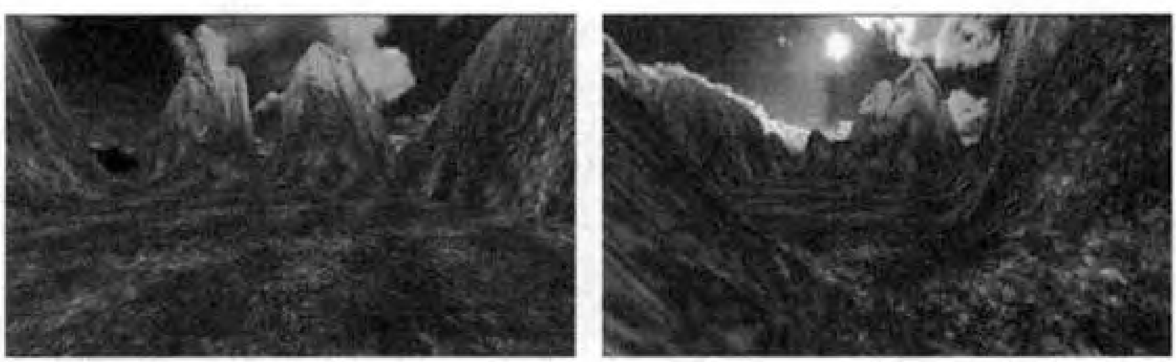
天空穹技术中不再是用立方体模拟天空，而是用一个半球面模拟天空，此半球面上需要贴上对应天空的纹理图，具体情况如图11-16所示。



图11-16　天空穹原理图

> **提示**
>
> 从图11-16中可以看出，用于模拟天空的半球面切分得越细效果就会越好。但实际开发中出于对性能的考量，切分得不宜过细。

了解了天空穹的原理后，就可以进行本小节案例Sample11_6的开发了。在介绍开发步骤之前请读者首先了解一下本案例的运行效果，如图11-17所示。


图11-17　天空穹案例效果图

> **提示**

从天空穹与天空盒案例效果图的对比中可以看出，天空穹的效果会更好一些。另外与Sample11_4相同，用手指按住屏幕左下角或右下角，摄像机会绕场景旋转；用手指按住屏幕左上角或右上角，摄像机会前进或后退，以从不同的位置、角度观察场景。

从图11-17中可以看出，本案例实际是对前面“MipMap地形”小节案例Sample11_4的升级，因此这里对相同的代码不再赘述，仅给出升级的主要步骤，具体如下所列。

（1）首先需要在案例中增加用于绘制天空穹的Sky类，此类其实就是一个半球面，大部分代码 与前面章节一些案例中纹理球面类（如第7章案例Sample7_4中的地球或月球类）的基本相同，这里不再赘述，需要的读者可以参考随书光盘中的源代码。

（2）由于本案例中应用于天空穹的纹理不是正方形（1024×256），同时升级前的纹理加载方法initTexture仅仅将纹理采样方式设置为MipMap系列的选项了。因此为了保证在大部分机型上能够正常运行本案例，需要对加载纹理的initTexture方法进行修改，修改后的代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_6/com/bn/Sample11_6目录下的MySurfaceView.java。

```
public int initTexture(int drawableId,boolean isMipmap)
{
    ……//此处省略了与升级前相同的代码，需要的读者请参考光盘
    if(isMipmap){//若标志位为 true，则将采样方式设置为 MipMap 系列
        GLES20.glTexParameteri ( GLES20.GL_TEXTURE_2D, //设置 MAG 纹理采样方式
          GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_LINEAR_MIPMAP_LINEAR);
        GLES20.glTexParameteri ( GLES20.GL_TEXTURE_2D, //设置 MIN 纹理采样方式
          GLES20.GL_TEXTURE_MIN_FILTER, GLES20.GL_LINEAR_MIPMAP_NEAREST);
    }else{      //若标志位为 false，则将采样方式设置为普通采样系列
            GLES20.glTexParameteri ( GLES20.GL_TEXTURE_2D, //设置 MAG 纹理采样方式
              GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_LINEAR);
            GLES20.glTexParameteri ( GLES20.GL_TEXTURE_2D, //设置 MIN 纹理采样方式
              GLES20.GL_TEXTURE_MIN_FILTER, GLES20.GL_NEAREST);
        }
        ……//此处省略了与升级前相同的代码，需要的读者请参考光盘
        if(isMipmap) {//若标志位为 true，则自动生成 MipMap 系列纹理
            GLES20.glGenerateMipmap(GLES20.GL_TEXTURE_2D);//自动生成 MipMap 系列纹理
        }
        ……//此处省略了与升级前相同的代码，需要的读者请参考光盘
    }
```

说明

从上述代码中可以看出，主要是增加了isMipmap入口参数，根据此参数的值将纹理采样方式设置为Mipmap系列或普通系列。同时，还根据此参数的值决定是否自动生成MipMap系列纹理。

（3）在MySurfaceView类中增加加载天空穹纹理、创建天空穹（Sky类）对象、绘制天空穹的相关代码。这些代码与本书前面很多案例的套路完全相同，这里不再赘述，需要的读者请参考随书光盘中的源代码。

11.3.3　天空盒与天空穹的使用技巧

前面两个小节给出的两个案例，第一个仅包含天空盒本身，第二个用天空穹罩住了不太大的山地地形。但在很多实际应用中场景非常大，若使用尺寸足以包含整个场景的天空盒或天空穹效果就不是很好。因此实际开发时经常会仅让天空盒或天空穹罩住场景的一部分，随着摄像机的移动天空盒或天空穹也跟着一起移动，具体情况如图11-18所示。

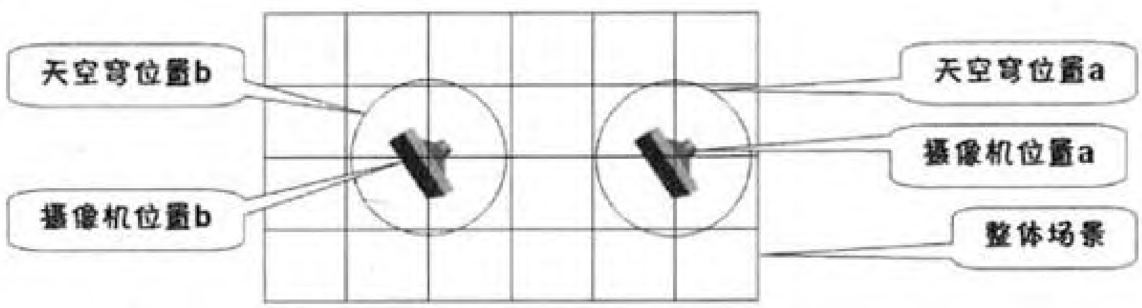


图11-18　天空盒/穹伴随摄像机移动示意图

采用这种天空盒或天空穹使用技巧的游戏有很多，如大名鼎鼎的极品飞车、都市赛车等，读者可以在玩这些游戏进行娱乐的同时细致观察一下就能感觉到。同时本书最后的大案例BN赛艇也是 采用此技术的，有兴趣的读者可以重点参考BN赛艇中使用天空穹的代码。

11.4　镜像技术

现实世界中，水边的树木、山体经水面反射后会形成倒影，这种效果在物理学中称之为镜像。不但水面可以形成镜像，一切表面光滑，能良好反射光线的物体都可以形成镜像。因此在开发很多模拟现实世界的3D场景中，若能够真实地再现镜像，吸引力将大大增加。本

节将向读者介绍一种实现镜像的技术，主要包括此技术的原理及一个篮球被地板反射形成镜像的案例。

11.4.1 镜像基本原理

从初中物理中大家都学到过，形成镜像的原因是反射，经过反射形成的像与其对应的实体相对于反射面是对称的。因此，在OpenGL ES中开发镜像效果时，最为关键的一步是根据实体的位置及反射面的位置朝向计算出像的位置，如图11-19所示。

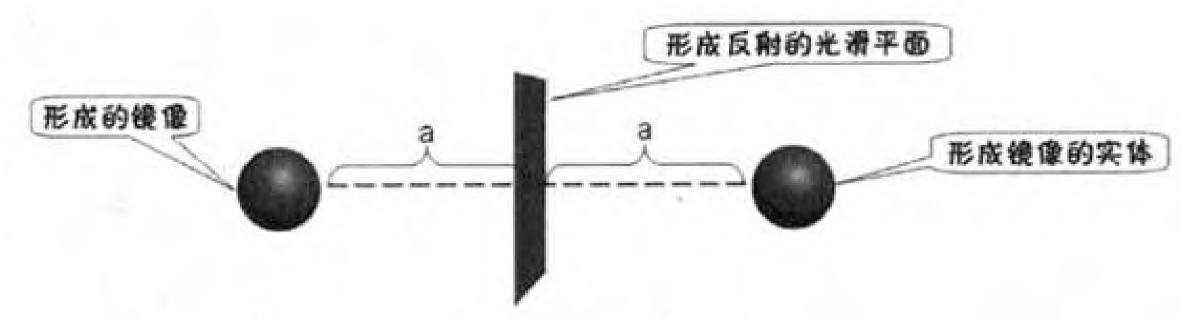


图11-19　镜像技术原理

> **提示**
>
> 开发中如果情况允许应该尽量让反射面平行于某个坐标平面（如XOY平面、XOZ平面、YOZ平面），这样像位置的计算就非常简单了。

11.4.2 基本效果案例

上一小节介绍了镜像技术的基本原理，本小节将给出一个简单的实现了镜像技术的案例Sample11_7，其运行效果如图11-20所示。

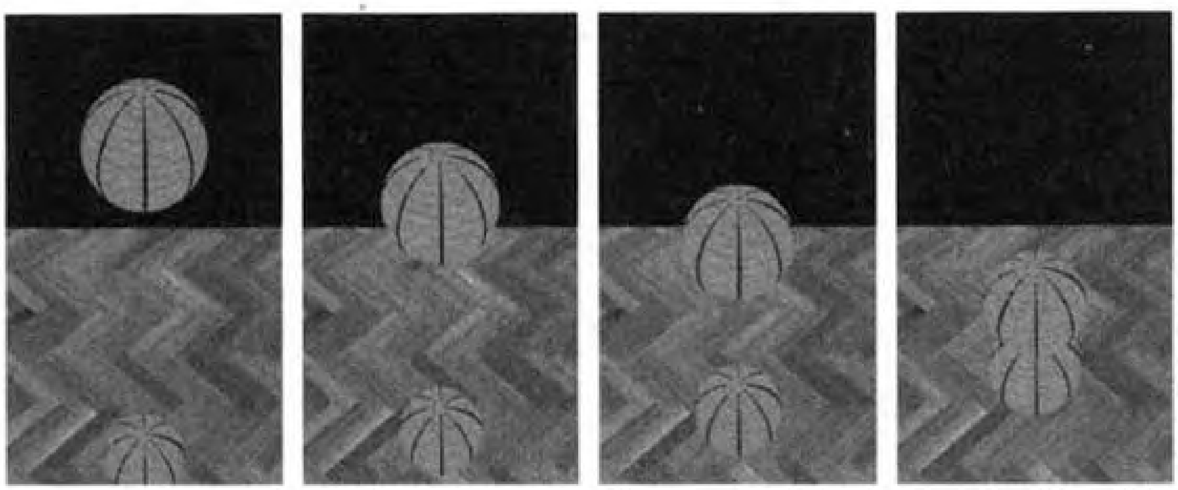


图11-20　运行效果图

说明

图11-20中呈现的是篮球被光滑木地板反射形成镜像的场景，从左至右依次为篮球开始下落，篮球继续下落、篮球碰到地板的几种情况。

了解了案例的运行效果后，就可以进行案例的开发了。由于本案例中用到的很多类与前面许多 案例的基本一致，因此这里仅介绍本案例中有特色的部分。主要包括绘制场景的MySurfaceView类以及完成篮球自由下落与反弹物理计算的BallForControl类，具体内容如下所列。

（1）首先介绍完成篮球自由下落与反弹物理计算的BallForControl类，其代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_7/com/bn/Sample11_7目录下的BallForControl.java。

```java
package com.bn.Sample11_7;                         //声明包名
import static com.bn.Sample11_7.Constant.*;        //引入常量类
public class BallForControl{
    public static final float TIME_SPAN=0.05f; //物理模拟的单位时间间隔
    public static final float G=0.8f;            //重力加速度
    BallTextureByVertex btv;                     //用于绘制篮球的纹理球
    float startY;                                //每轮起始点位置
    float timeLive=0;                            //此周期存活时长
    float currentY=0;                            //当前 Y 位置
        float vy=0;                              //每轮初始 Y 轴方向速度
        public BallForControl(BallTextureByVertex btv,float startYIn){
            this.btv=btv; this.startY=startYIn; currentY=startYIn;//初始化速度与起始位置
            new Thread(){                        //开启一个线程运动篮球
              public void run(){
                while(true){
                  timeLive+=TIME_SPAN;           //此轮运动时间增加
                  //根据此轮起始 Y 坐标、此轮运动时间、此轮起始速度计算当前位置
                  float tempCurrY=startY-0.5f*G*timeLive*timeLive+vy*timeLive;
                  if(tempCurrY<=0){              //若当前位置低于地面则碰到地面反弹
                          startY=0;              //反弹后起始高度为 0
                          vy=-(vy-G*timeLive)*0.8f; //反弹后起始速度
                          timeLive=0;            //反弹后此轮运动时间清 0
                          if(vy<0.35f){          //若速度小于阈值则停止运动
                              currentY=0;
                              break;
                      }}else{//若没有碰到地面则正常运动
                          currentY=tempCurrY;
                      }
                      try{Thread.sleep(20);}catch(Exception e){e.printStack
                      Trace();}//线程休眠
```

```
30                }}}.start();
31         }
32         public void drawSelf(int texId){    //绘制物体自身
33             MatrixState.pushMatrix();      //保护现场
34             MatrixState.translate(0, UNIT_SIZE*BALL_SCALE+currentY, 0);  //平移变换
35             btv.drawSelf(texId);           //绘制篮球
36             MatrixState.popMatrix();       //恢复现场
37         }
38         public void drawSelfMirror(int texId){ //绘制镜像体
39             MatrixState.pushMatrix();      //保护现场
40             MatrixState.translate(0, -UNIT_SIZE*BALL_SCALE-currentY, 0); //平移变换
41             btv.drawSelf(texId);           //绘制篮球
42             MatrixState.popMatrix();       //恢复现场
43         }
44     }
```

- 上述代码主要是实现了上抛运动的物理定律，其中第21行的0.8f为恢复系数，其代表每次篮球碰撞地面后还能保存的动能占碰撞前动能的比例。
- 第32-37行的drawSelf方法根据篮球当前的*Y*坐标绘制篮球，第38-43行的drawSelfMirror方法根据篮球的*Y*坐标计算镜像体位置并绘制篮球。

（2）接着介绍绘制整体场景的MySurfaceView类，其代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_7/com/bn/Sample11_7目录下的MySurfaceView.java。

```
package com.bn.Sample11_7;                                    //声明本案例包名
......//此处省略了部分类的导入代码，读者可以自行查阅随书光盘中的源代码
class MySurfaceView extends GLSurfaceView{
    private SceneRenderer mRenderer;                          //场景渲染器
    int textureFloor;                                         //木地板纹理 id
    int textureBallId;                                        //篮球纹理 id
    ......//此处省略了本类中构造器的代码，读者可以自行查阅随书光盘中的源代码
    private class SceneRenderer implements GLSurfaceView.Renderer{
        TextureRect texRect;                                  //表示地板的纹理矩形
            BallTextureByVertex btbv;                         //用于绘制的纹理球
            BallForControl bfd;                               //完成物理计算球控制类
            public void onDrawFrame(GL10 gl){                 //绘制一帧画面的方法
             GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT|
                GLES20.GL_COLOR_BUFFER_BIT);                  //清除深度缓冲与颜色缓冲
             MatrixState.pushMatrix();                        //保护现场
             MatrixState.translate(0, -2, 0);                 //平移变换
             texRect.drawSelf(textureFloor);                  //绘制木地板
             bfd.drawSelfMirror(textureBallId);               //绘制镜像体
             bfd.drawSelf(textureBallId);                     //绘制实际物体
             MatrixState.popMatrix();                         //恢复现场
            }
            ......//此处省略了 onSurfaceChanged 方法，读者可以自行查阅随书光盘中的源代码
            public void onSurfaceCreated(GL10 gl, EGLConfig config){
             GLES20.glClearColor(0.0f,0.0f,0.0f,1.0f); //设置屏幕背景色 RGBA
```

```
25            texRect=new TextureRect(MySurfaceView.this,4,2.568f);//创建纹理矩形对象
26             //创建用于绘制篮球的纹理球对象
27            btbv=new BallTextureByVertex(MySurfaceView.this,BALL_SCALE);
28             //创建用于完成篮球自由下落与反弹物理计算的 BallForControl 类对象
29            bfd=new BallForControl(btbv,3f);
30            GLES20.glDisable(GLES20.GL_DEPTH_TEST);           //关闭深度检测
31             ......//此处省略了一些纹理加载、打开背面剪裁、初始化变换矩阵的代码
32      }}
33      ....//此处省略了加载纹理的 initTexture 方法，读者可以自行查阅随书光盘中的源代码
34  }
```

- 第12-21行为绘制一帧画面的onDrawFrame方法，其中首先绘制了木地板（反射面），然后绘制了镜像体与实际物体。镜像体与实际物体位置的计算是由BallForControl类完成的，前面步骤中已经介绍。
- 第23-32行为进行初始化的onSurfaceCreated方法，其中大部分代码与前面很多案例的套路完全一致，最大的区别就是别的案例几乎都是打开深度检测的，而这里是关闭深度检测。

提示

实现镜像效果时除了需要计算出镜像的位置，最关键的就是需要关闭深度检测。若绘制镜像时不关闭深度检测，则距离摄像机较近的反射面就会挡住镜像体，如图11-21所示，这点也请读者在开发中特别注意。

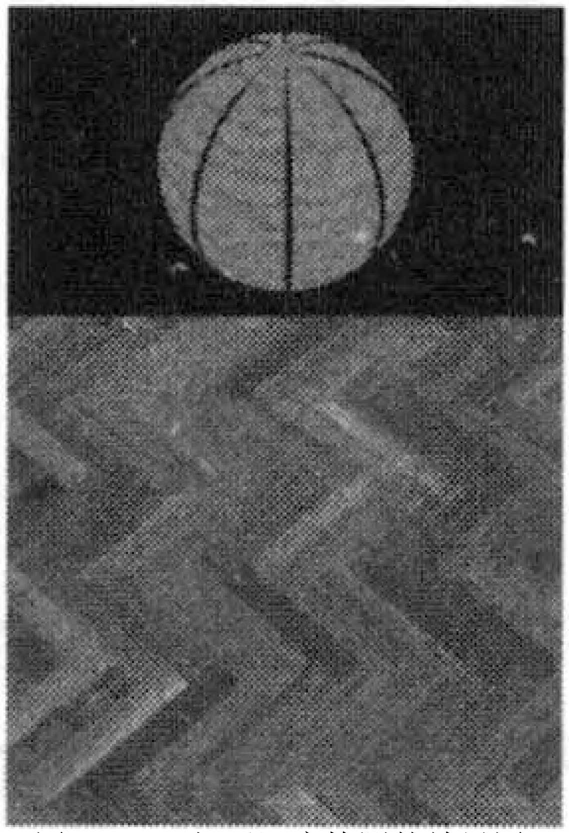


图11-21　打开深度检测的效果图

之所以打开深度检测就看不见镜像体原因如下。

• 若先绘制镜像体，则有镜像体片元的位置深度缓冲区中会记录较大的深度，再绘制距离较近的反射面时，其片元对应的深度较小，深度检测通过，将覆盖原有的片元，如图11-22（a）所示。

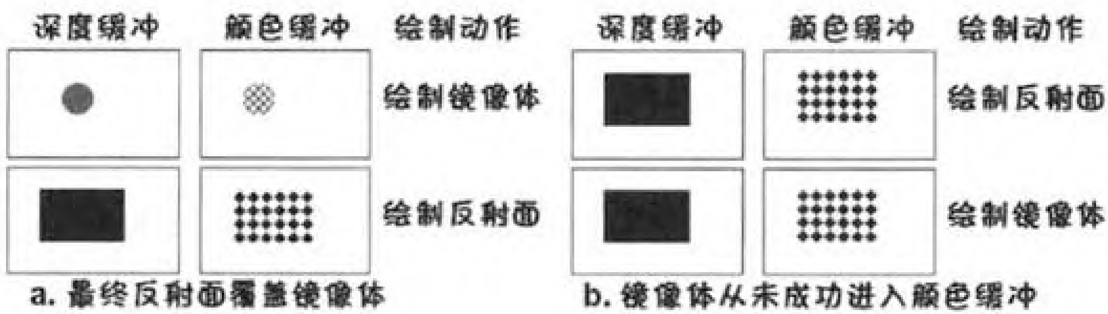


图11-22　绘制镜像体时需要关闭深度检测的原因

• 若先绘制反射面，则有反射面片元的位置深度缓冲区中会记录较小的深度，再绘制距离较远的镜像体时，其片元对应的深度大，深

度检测失败，片元将被丢弃，如图11-22（b）所示。

> **提示**
>
> 图11-22中颜色越深表示深度越小，这是因为从灰度值的角度考虑值越大颜色越浅。最后需要读者注意的是，即使关闭了深度检测，绘制顺序也不是任意的，必须先绘制反射面，再绘制镜像体。

11.4.3　升级效果案例

上一小节案例中实现的镜像场景已经较为真实，但还有一个小的瑕疵。那就是镜像与实体是一 模一样的，这不完全符合现实世界的情况。现实世界中，镜像会受到反射面的影响，也就是镜像上会隐约映射出反射面的内容，不应该与实体一模一样。

本小节将上一小节的案例进一步升级（升级后的案例为Sample11_8），实现上述真实镜像的效果，如图11-23所示。

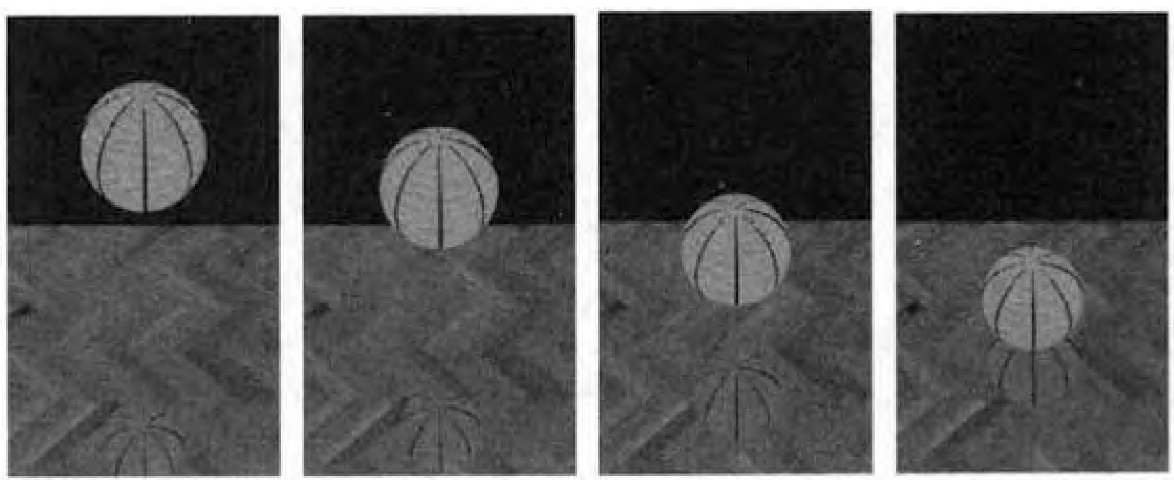


图11-23　升级后的运行效果图

从图11-23中可以看出，镜像上隐约映射出了反射面的内容，真实感比升级前提高了不少。实现的思路也很简单，只需要绘制完镜像体之后再采用混合模式绘制一个半透明的反射面即可。

了解了实现思路后，就可以进行案例的开发了。由于实际上本案例仅仅是将案例Sample11_7复制了一份并进行了修改，因此这里仅给出修改的主要步骤。

（1）首先需要在项目中增加一幅半透明的木地板纹理图mdbtm.png，其内容与原有的不透明的木地板纹理图完全一致。

（2）在MySurfaceView类中增加加载半透明纹理以及绘制半透明反射面的代码，修改后绘制场景的onDrawFrame方法如下。

代码位置：见随书光盘中源代码/第11章/Sample11_8/com/bn/Sample11_8目录下的MySurfaceView.java。

```
public void onDrawFrame(GL10 gl){  //实现GLSurfaceView.Renderer接口后重写的绘制方法
   GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT|
        GLES20.GL_COLOR_BUFFER_BIT);                //清除深度缓冲与颜色缓冲
   MatrixState.pushMatrix();                        //保护现场
   MatrixState.translate(0, -2, 0);                 //平移变换
   texRect.drawSelf(textureFloor);                  //绘制反射面地板（不透明）
   bfd.drawSelfMirror( textureBallId);              //绘制镜像体
   GLES20.glEnable(GLES20.GL_BLEND);                //开启混合
   GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA,
            GLES20.GL_ONE_MINUS_SRC_ALPHA);         //设置混合因子
      texRect.drawSelf(textureFloorBTM);            //绘制半透明地板
      GLES20.glDisable(GLES20.GL_BLEND);            //关闭混合
      bfd.drawSelf(textureBallId);                  //绘制实际物体
      MatrixState.popMatrix();                      //恢复现场
  }
```

说明

请读者要特别注意绘制的顺序，先绘制不透明反射面，接着绘制镜像体，再绘制半透明反射面，最后绘制实际物体，不正确的顺序可能不能产生正确的效果。

11.5　动态文本输出

到目前为止，本书已经给出了很多酷炫的案例，但这些案例场景中都没有出现文字。这是因为OpenGL ES 2.0本身并没有提供在3D场景中绘制文字的方法，要想在场景中看到文字只能将文字先存储到纹理图中，再应用有文字内容的纹理图才可以。

通过上面的描述读者可能会想到一个问题，纹理图中的文字是固定不变的啊，如果需要动态内容的文字怎么办呢？本节将通过一个案例向读者介绍如何在3D场景中呈现动态内容的文本。

11.5.1　案例效果与基本原理

介绍本案例的具体开发步骤之前，首先需要了解一下本案例的运行效果与基本原理，其运行效果如图11-24所示。



图11-24　动态文本案例运行效果

> **说明**
>
> 图11-24中从左至右为文本从上往下动态播放的情况，动态播放的文本内容为唐代著名诗人李白的侠客行。

本案例的实现原理也不复杂，那就是在绘制每帧画面前首先调用2D的API生成带本帧画面所需文本内容的纹理图，然后再将此幅纹理图

加载并通过纹理矩形呈现出来，整个过程如图11-25所示。

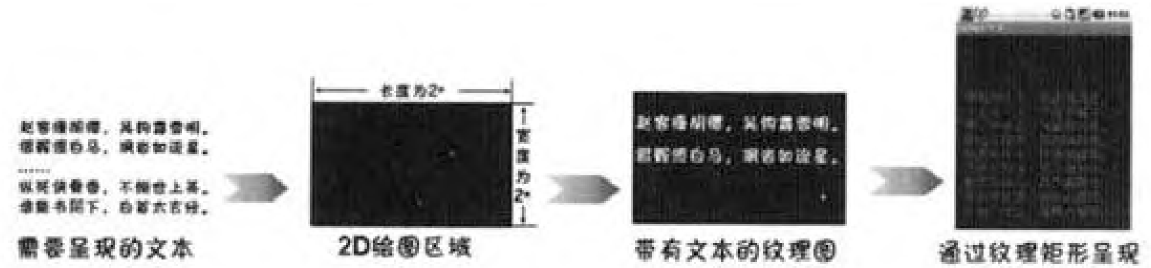


图11-25　动态文本输出过程示意图

11.5.2　具体开发步骤

了解了案例的基本原理与运行效果后，就可以进行案例的开发了。由于本案例中3D相关的大部分代码与前面很多案例中的基本相同，因此这里仅给出有代表性的部分，具体如下所列。

（1）首先介绍的是生成带有所需文本内容纹理图的工具类FontUtil，其代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_9/com/bn/Sample11_9目录下的FontUtil.java。

```
package com.bn.Sample11_9;                                    //声明包名
......//此处省略了部分类的导入代码，读者可以自行查阅随书光盘中的源代码
public class FontUtil{
    static int cIndex=0;                                      //文本内容索引
    static final float textSize=40;                           //字体的大小
    static int R=255;                                         //画笔红色通道的值
    static int G=255;                                         //画笔绿色通道的值
    static int B=255;                                         //画笔蓝色通道的值
    public static Bitmap generateWLT(String[] str,int width,int height){
                                                              //生成文本纹理图的方法
            Paint paint=new Paint();                          //创建画笔对象
            paint.setARGB(255, R, G, B);                      //设置画笔颜色
            paint.setTextSize(textSize);                      //设置字体大小
            paint.setTypeface(null);
            paint.setFlags(Paint.ANTI_ALIAS_FLAG);            //打开抗锯齿，使字体边缘光滑
            Bitmap bmTemp=Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_8888);
            Canvas canvasTemp = new Canvas(bmTemp);           //根据指定的位图创建画布
            for(int i=0;i<str.length;i++){                    //绘制当前纹理图对应的每行文本
                canvasTemp.drawText(str[i], 0, textSize*i+(i-1)*5, paint);
            }
            return bmTemp;                                    //返回绘制的作为纹理图的位图
        }
        static String[] content={                             //所有内容所属的字符串数组
            "赵客缦胡缨，吴钩霜雪明。",
            ......//此处省略了本数组中的部分内容，读者可以自行查阅随书光盘中的源代码
            "谁能书阁下，白首太玄经。",
        };
        public static String[] getContent(int length,String[] content){
                                                              //获取指定行数字符串数组的方法
            String[] result=new String[length+1];             //创建字符串数组
            for(int i=0;i<=length;i++){
                result[i]=content[i];                         //将当前需要的内容填入数组
            }
            return result;
        }
        public static void updateRGB(){                       //随机产生画笔颜色值的方法
            R=(int)(255*Math.random());                       //随机产生画笔红色通道值
            G=(int)(255*Math.random());                       //随机产生画绿红色通道值
            B=(int)(255*Math.random());                       //随机产生画蓝红色通道值
    } }
```

- 第9-21行为本类的核心方法generateWLT，其将指定字符串数组中的内容绘制到位图上。 此方法绘制的位图后面将作为纹理图通过纹理矩形呈现到屏幕上。
- 第27-33行为根据传入的参数动态产生一帧画面中所需文本内容的getContent方法，此方法根据应用需求的不同会有所变化。本案例中采用的是将指定字符串数组中需要的一部分截取出来的策略。

> **提示**
>
> 第15行创建位图对象时采用的格式是ARGB_8888，还有一些其他可选的格式，如ALPHA_8、ARGB_4444以及RGB_565，需要的时候可以选用。

（2）接着需要介绍的是绘制场景的MySurfaceView类，其代码如下。

代码位置：见随书光盘中源代码/第11

章/Sample11_9/com/bn/Sample11_9目录下的MySurfaceView.java。

```java
package com.bn.Sample11_9;                                          //声明包名
......//此处省略了部分类的导入代码，读者可以自行查阅随书光盘中的源代码
public class MySurfaceView extends GLSurfaceView{
    SceneRenderer mRender;                                          //渲染器引用
    ......//此处省略了本类的构造器，读者可以自行查阅随书光盘中的源代码
    private class SceneRenderer implements GLSurfaceView.Renderer{    //渲染器类
        TextRect tRect;                                             //纹理矩形
        int wlWidth=512;                                            //文本纹理图的宽度
        int wlHeight=512;                                           //文本纹理图的高度
            long timeStamp=System.currentTimeMillis();              //获取当前系统时间
            int texId=-1;                                           //当前的纹理 id
            @Override
            public void onDrawFrame(GL10 gl){                       //绘制一帧画面的方法
              long tts=System.currentTimeMillis();                  //获取绘制此帧时的系统时间
              if(tts-timeStamp>500){ //判断时间差是否大于 500ms，如果大于 500ms 则更新文本
                    timeStamp=tts;                                  //记录更新的时间
                    FontUtil.cIndex=(FontUtil.cIndex+1)%FontUtil.content.length;
                    FontUtil.updateRGB();                           //重新生成画笔颜色
                }
                if(texId!=-1){                                      //若当前纹理存在，则删除
                    GLES20.glDeleteTextures(1, new int[]{texId}, 0);
                }
                Bitmap bm=FontUtil.generateWLT(                     //重新生成文本纹理图
                 FontUtil.getContent(FontUtil.cIndex, FontUtil.content),wlWidth, wlHeight);
                texId=initTexture(bm);                              //将生成的纹理图加载
                GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT|
                    GLES20.GL_COLOR_BUFFER_BIT);                    //清除深度缓冲与颜色缓冲
                MatrixState.pushMatrix();                           //保护现场
                MatrixState.translate(0, 0, -2);                    //平移变换
                tRect.drawSelf(texId);                              //绘制纹理矩形
                MatrixState.popMatrix();                            //恢复现场
            }
            ......//此处省略了重写的 onSurfaceChanged 方法与 onSurfaceCreated 方法
            ......//需要的读者可以自行查阅随书光盘中的源代码
    }
```

```
        ……//此处省略了加载纹理图的initTexture方法，与前面很多案例中的基本相同
}
```

说明

上述MySurfaceView类的大部分代码与前面很多案例中的相同，区别较大的就：是此案例中纹理图是每绘制一帧时单独加载的，而且在加载新的纹理前首先会删除上一帧所用的纹理。这是因为此案例中的纹理是动态产生的，所以要在绘制时加载；另外纹理存储是有限的，因此加载新的之前一定要删除旧的，否则运行一段时间程序就会崩溃。

11.6　非真实感绘制

到目前为止，本书所给出的案例都是以逼真为最高追求的，但这仅仅是一个方面。有时希望程序能够绘制出不太真实的效果，如水彩画、水粉画等。这就是所谓的非真实感绘制，本节将通过一个案例带领读者简单了解一下非真实感绘制效果的开发。

11.6.1　案例效果与基本原理

介绍本案例的具体开发步骤之前，首先需要了解一下本案例（Sample11_10）的运行效果与基本原理，其运行效果如图11-26所示。



图11-26　非真实感绘制案例的运行效果

提示：运行本案例时用手指在屏幕上滑动可以旋转场景中的物体。

从图11-26中可以看出，本节非真实感绘制案例实现的是水粉画效果，也称之为卡通着色。卡通着色主要是使用不变的颜色与明显的轮廓来对物体进行着色，其主要策略是减少颜色的变化。使用卡通着色时首先要计算光照，然后对散射光与镜面光通道采用特殊的处理以得到最终片元的颜色，具体情况如下所列。

- 散射光通道着色。

散射光通道着色只需要两个颜色值，一个用来表示明亮区域，另一个表示昏暗区域。具体说就是根据光照计算得出的散射光通道最终强度来确定当前片元属于哪个区域，然后采用此区域对应的颜色对片元着色。从数学角度来说，就是将颜色的取值从连续函数转化为阶梯函数，如图11-27所示。

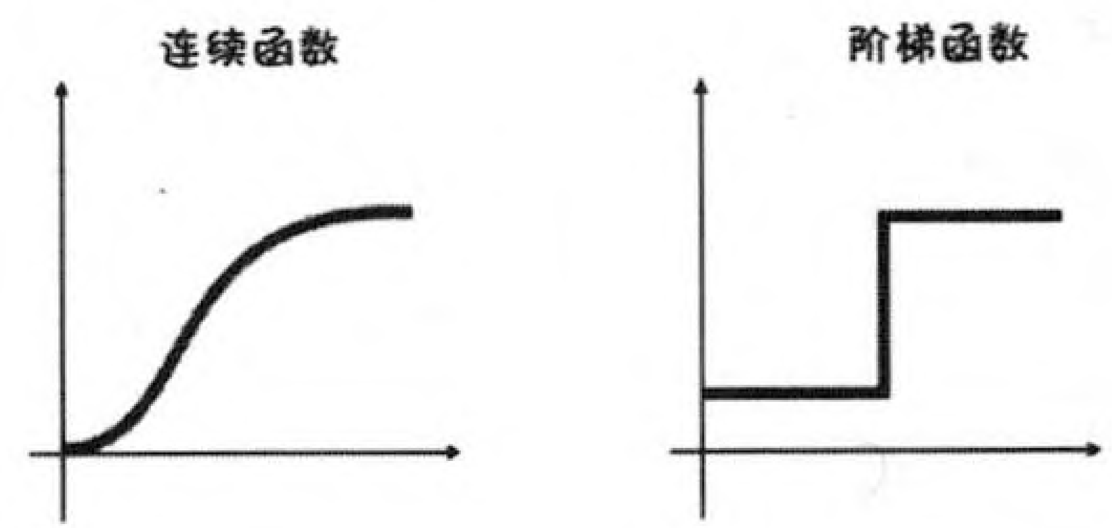


图11-27　连续函数与阶梯函数

- 镜面光通道着色。

镜面光通道着色的策略与散射光通道着色类似，也是将连续取值的函数转化为阶梯函数。稍有不同的是，镜面光通道着色中昏暗区域不着色（此情况下颜色各通道值都为0）。

- 轮廓描绘。

除了上述散射光通道与镜面光通道的着色，卡通着色还有一项很重要的工作就是对物体边缘进行描绘（通常采用黑色）。而描绘边缘前，最重要的就是要确定当前着色的片元是否处于边缘。判断片元是否处于边缘的策略非常简单，那就是考量视线向量与此片元处法向量的夹角，若夹角大于一定的值则认为是边缘。实际开发时只需要通过两个向量的点积值进行判断即可。

11.6.2　具体开发步骤

上一小节介绍了本案例的运行效果与基本原理，下面将介绍案例的具体开发步骤，具体如下所列。

（1）用3DMax设计如图11-26所示的模型并导出成obj文件放入项目的assets目录待用。

（2）开发出搭建场景的基本代码，包括：加载物体、摆放物体、计算光照等。这些代码与前面章节的许多案例基本套路完全一致，因此这里不再赘述。

（3）开发为卡通着色服务的特色着色器，首先是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_10/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                               //总变换矩阵
uniform mat4 uMMatrix;                                 //变换矩阵
uniform vec3 uLightLocation;                           //光源位置
uniform vec3 uCamera;                                  //摄像机位置
attribute vec3 aPosition;                              //顶点位置
attribute vec3 aNormal;                                //顶点法向量
varying vec4 diffuse;                                  //用于传递给片元着色器的散射光最终强度
varying vec4 specular;                                 //用于传递给片元着色器的镜面光最终强度
varying float vEdge;                                   //视线向量与法向量的点积值
  void pointLight(                                     //定位光光照计算的方法
    in vec3 normal,                                    //法向量
    inout vec4 diffuse,                                //散射光最终强度
    inout vec4 specular,                               //镜面光最终强度
    inout float edge,                                  //视线向量与法向量的点积值
    in vec3 lightLocation,                             //光源位置
    in vec4 lightDiffuse,                              //散射光强度
    in vec4 lightSpecular                              //镜面光强度
  ){
    vec3 normalTarget=aPosition+normal;                //计算变换后的法向量
    vec3 newNormal=(uMMatrix*vec4(normalTarget,1)).xyz-(uMMatrix*vec4(aPosition,1)).xyz;
    newNormal=normalize(newNormal);                    //对法向量规格化
    vec3 eye= normalize(uCamera-(uMMatrix*vec4(aPosition,1)).xyz);
                                                       //计算从表面点到摄像机的向量
    edge = max(0.0,dot(newNormal,eye));                //计算视线向量与法向量的点积值
    ……//此处省略了计算散射光和镜面反射光最终确定的代码，需要时请参考光盘
  }
  void main() {
    gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点的位置
    float edge;
    pointLight(normalize(aNormal),diffuse,specular,edge,   //计算光照和边缘信息
        uLightLocation,vec4(0.7,0.7,0.7,1.0),vec4(0.3,0.3,0.3,1.0));
```

```
    vEdge = edge;                    //将视线向量与法向量的点积值传入片元着色器
}
```

说明

从上述顶点着色器的代码中可以看出，与普通的光照着色器基本相同，主要的区别是增加了计算视线向量与法向量的点积值并传入片元着色器的相关代码。

（4）开发完顶点着色器后，就可以开发片元着色器了，其代码如下。

代码位置：见随书光盘中源代码/第11章/Sample11_10/asset目录下的frag.sh。

```
precision mediump float;                //给出默认的浮点精度
varying vec4 diffuse;                   //接收从顶点着色器传递过来的散射光最终强度
varying vec4 specular;                  //接收从顶点着色器传递过来的镜面光最终强度
varying float vEdge;                    //接收从顶点着色器传递过来的视线向量与法向量的点积值
void main(){
  float averageDiffuse = (diffuse.x + diffuse.y + diffuse.z)/3.0;
                                        //计算散射光3个色彩通道的平均值
  if(averageDiffuse<0.5){               //散射光强度平均值小于0.5时，设置该值为0.2
    averageDiffuse=0.2;
  } else {                              //散射光强度平均值为其他情况时，设置该值为0.8
       averageDiffuse=0.8;
     }
     //生成用于卡通着色的散射光强度值
     vec4 diffuseFinal = vec4(averageDiffuse,averageDiffuse,averageDiffuse,1.0);
     //计算镜面光3个色彩通道的平均值
     float averageSpecular = (specular.x + specular.y + specular.z)/3.0;
     if(averageSpecular<0.18){          //镜面光强度平均值小于0.18时，设置该值为0.0
       averageSpecular=0.0;
     } else {                           //镜面光强度平均值不小于0.18时，设置该值为1.0
       averageSpecular=1.0;
     }
     //生成用于卡通着色的镜面光强度值
     vec4 specularFinal = vec4(averageSpecular, averageSpecular, averageSpecular, 1.0);
     float edgeFinal = 1.0;             //不位于边缘的片元的颜色值系数
     if(vEdge<0.2){//若小于0.2，则说明视线向量与法向量的夹角很大，接近90°，片元位于边缘
        edgeFinal = 0.0;                //位于边缘的片元的颜色值系数
     }
     gl_FragColor = edgeFinal*(specularFinal+diffuseFinal);//加权计算出片元的最终颜色值
  }
```

说明

上述片元着色器中有代表性的是，根据光照计算的结果实现了上一小节中介绍的卡通着色算法，其他的部分与普通的光照着色器基本相同。

11.7 本章小结

本章主要介绍了一些常用的3D场景开发技巧，主要包括标志板、灰度图地形、天空盒、天空穹、镜像、动态文本输出以及非真实感绘制等。掌握了这些技术以后，读者开发3D场景的能力应该大大增强，可以开发出更加酷炫的应用。

第12章 几种剪裁与测试

玩过游戏的读者都知道，很多游戏运行时，屏幕上不单显示主视角游戏场景，还可能会显示仪表板或次视角游戏场景。要想满足这样的需求仅仅采用前面章节介绍的技术几乎是不可能的，因此本章将向读者介绍一些能满足特殊需要的剪裁与测试方面的知识，主要包括：剪裁测试、Alpha测试、模板测试、任意剪裁平面。

12.1 剪裁测试

剪裁测试主要用来在渲染场景时限制绘制区域，用其可以方便地实现同时在屏幕上绘制主视角与次视角场景。本节将首先介绍剪裁测

试的基本原理，然后再给出一个具体的案例Sample12_1。

12.1.1　基本原理与核心代码

前面已经提过，剪裁测试可以在渲染时用来限制绘制区域，通过此技术可以在屏幕（帧缓冲）上指定一个矩形区域。启用剪裁测试后，绘制将不会像前面章节介绍过的案例一样在整个屏幕（帧缓冲）中进行，而是仅在指定的矩形区域中进行。

不在此矩形区域内的片元将被丢弃，只有在此矩形区域内的片元才有机会最终进入帧缓冲。因此实际达到的效果就是在屏幕上开辟了一个小窗口，可以在其中进行特定内容的绘制。

> **提示**
>
> 很多读者可能玩过《魔兽争霸3》这款游戏，玩游戏时如果选中一个士兵，则屏幕下方的一个小方框内就会出现该士兵的动态的立体头像。为了保证该头像无论如何都不会因越界而覆盖到外面的内容，此时就可以使用剪裁测试。

使用剪裁测试所需要的核心代码如下所列。

```
1 GLES20.glEnable(GL10.GL_SCISSOR_TEST);          //启用剪裁测试
2 GLES20.glScissor(0,0,100,200);                  //设置区域
3 GLES20.glDisable(GL10.GL_SCISSOR_TEST);         //禁用剪裁测试
```

> **提示**
>
> 第2行glScissor方法中前两个参数为剪裁区域左下角的*X*、*Y* 坐标，后两个参数为剪裁区域的宽度和高度，单位为像素。

12.1.2　一个主次视角的简单案例

上一小节介绍了剪裁测试的基本原理与核心代码，本小节将通过一个简单的案例来介绍剪裁测试在实际开发中的应用，其运行效果如图12-1所示。

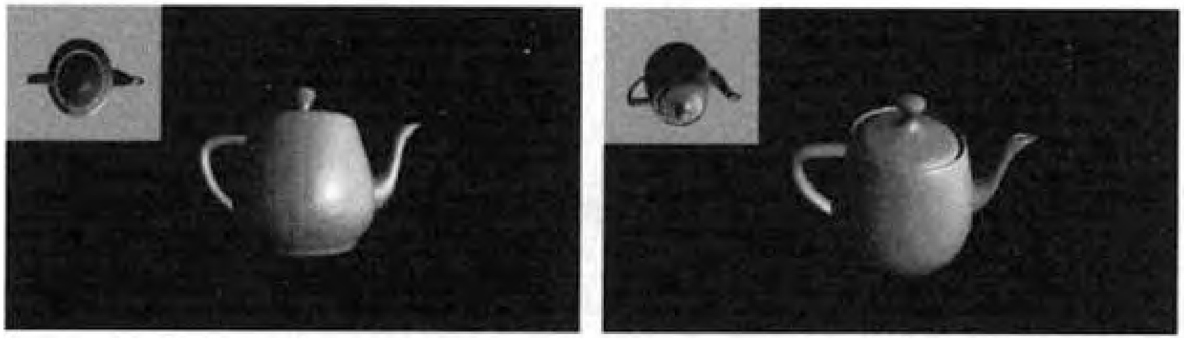


图12-1　案例Sample12_1的运行效果图

了解了本小节案例的运行效果后，下面将详细介绍本案例的开发。由于本案例中的大部分代码与第9章案例Sample9_3中的相同，因此这里仅给出本案例中有代表性的通过剪裁测试在屏幕上同时绘制主次视角场景的代码，具体内容如下。

代码位置：见随书光盘中源代码/第12章/Sample12_1/com/bn/Sample12_1目录下的MySurfaceView.java。

```
private class SceneRenderer implements GLSurfaceView.Renderer {   //场景渲染器类
    float yAngle;                                          //绕 Y 轴旋转的角度
    float zAngle;                                          //绕 Z 轴旋转的角度
    LoadedObjectVertexNormal lovo;                         //从指定的 obj 文件中加载的对象
    public void onDrawFrame(GL10 gl) {
        GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);       //设置屏幕背景色 RGBA
        GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT//清除深度缓冲与颜色缓冲
                | GLES20.GL_COLOR_BUFFER_BIT);
        //下述代码为绘制主视角场景服务-----------------------------------------------
            MatrixState.setProjectFrustum(-ratio, ratio, -1, 1, 2, 100);//设置投影矩阵
            MatrixState.setCamera(0, 0, 0, 0f, 0f, -1f, 0f, 1.0f, 0.0f);
                                                           //设置摄像机观察矩阵
            MatrixState.pushMatrix();                      //保护现场
            MatrixState.translate(0, -2f, -25f);           //平移变换
            MatrixState.rotate(yAngle, 0, 1, 0);           //绕 Y 轴旋转变换
            MatrixState.rotate(zAngle, 1, 0, 0);           //绕 Z 轴旋转变换
            if (lovo != null) {                            //绘制主视角场景中的物体
                lovo.drawSelf();
            }
            //下述代码为绘制此视角场景服务-------------------------------------------
            GLES20.glEnable(GL10.GL_SCISSOR_TEST);         //启用剪裁测试
            GLES20.glScissor(0, 480 - 200, 230, 200);      //设置剪裁区域
            GLES20.glClearColor(0.7f, 0.7f, 0.7f, 1.0f);   //设置屏幕背景色 RGBA
            GLES20.glClear(GL10.GL_COLOR_BUFFER_BIT//清除颜色缓存与深度缓存
                    | GL10.GL_DEPTH_BUFFER_BIT);
            //设置投影矩阵
            MatrixState.setProjectFrustum(-0.62f * ratio, 1.38f * ratio,-1.55f, 0.45f, 2, 100);
            //设置摄像机观察矩阵
            MatrixState.setCamera(0, 50f, -30f, 0, -2f, -25f, 0f, 0.0f, -1.0f);
```

```
            if (lovo != null) {                    //绘制次视角场景中的物体
                lovo.drawSelf();
            }
            GLES20.glDisable(GL10.GL_SCISSOR_TEST);//禁用剪裁测试
            MatrixState.popMatrix();               //恢复现场
        }
        ……//此处省略了onSurfaceChanged和onSurfaceCreated方法，需要时请查阅光盘
}
```

> **说明**
>
> 从上述代码中可以看出，当希望同时在屏幕上绘制两个不同视角的场景时，可以先绘制主视角场景，再启用剪裁测试设置好剪裁窗口，然后绘制次视角场景即可，整体的实现非常简单。

12.2 Alpha测试

通过上一小节剪裁测试的学习，读者可能会发现一个小问题：剪裁测试的区域只能是矩形。是的，通过剪裁测试不能实现任意形状的绘制区域，因此本节将向读者介绍一种可以实现任意形状绘制区域的技术——Alpha测试。

12.2.1 Alpha测试基本原理

通过前面章节的学习，读者应该已经知道OpenGL ES中每个颜色有4个色彩通道——RGBA，其中的A就是Alpha（透明度）通道。同时，读者也应该已经知道，在进行混合模式绘制时可以通过特定的混合因子使用Alpha通道值执行混合操作。

其实，Alpha通道值还有另外一个用途，那就是进行Alpha测试。Alpha测试的基本原理为，当绘制一个片元时，首先检测其Alpha值，

若Alpha值满足要求，则通过测试，绘制此片元，否则丢弃此片元，不进行绘制，具体情况如图12-2所示。

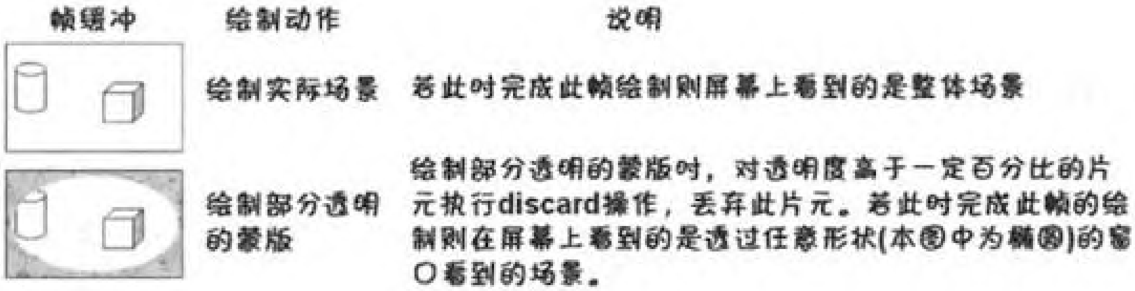


图12-2　Alpha测试的基本原理

提示

在OpenGL ES 1.x中有直接支持Alpha测试的API，但在OpenGL ES 2.0中这部分API被删除了。看到这里读者可能会担心：那怎么办？其实不用担心，由于本书介绍的是强大的可编程渲染管线，所以可以方便地通过自行开发着色器来实现Alpha测试，下一小节将要给出的案例就是如此。

12.2.2　一个椭圆窗口的案例

上一小节介绍了Alpha测试的基本原理，本小节将给出一个通过Alpha测试实现椭圆窗口的案 例Sample12_2，其运行效果如图12-3所示。



图12-3　案例Sample12_2的运行效果图

从图12-3中可以看出，案例呈现的是通过一个椭圆形窗口观察小树林的场景，其中用于实现椭圆窗口的部分透明纹理图如图12-4所示。

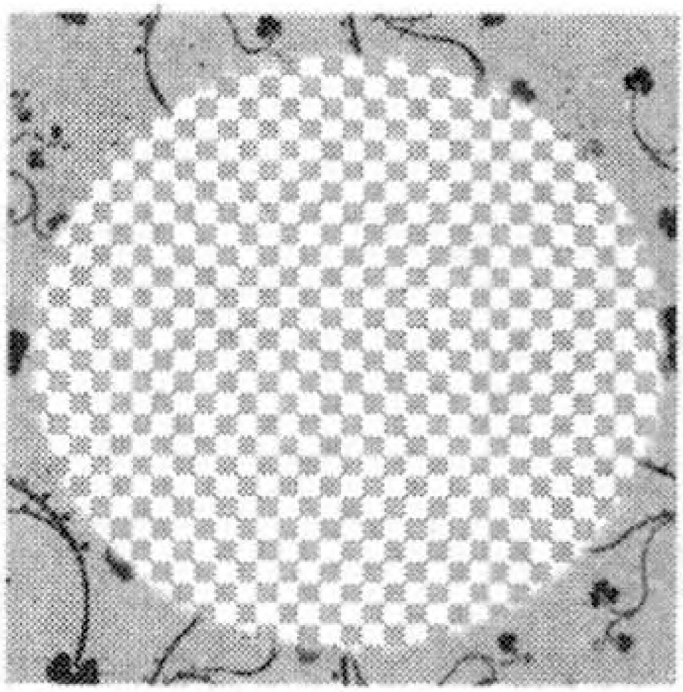


图12-4　椭圆窗口部分透明纹理

提示

图12-4中灰白相间的格子区域表示透明区域，这是一种约定俗成的表示方式。

了解了案例的运行效果后，接下来对本案例的具体开发过程进行介绍。由于本案例中很多类和前面章节案例中的很类似，因此在这里只给出本案例中特殊的、具有代表性的部分，具体内容如下。

（1）首先介绍MySurfaceView类中用于绘制一帧画面的onDrawFrame方法，其代码如下。

代码位置：见随书光盘中源代码/第12

章/Sample12_2/com/bn/Sample12_2目录下的MySurfaceView.java。

```
public void onDrawFrame(GL10 gl) {                                          //绘制一帧画面的方法
    GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT                               //清除深度缓冲与颜色缓冲
            | GLES20.GL_COLOR_BUFFER_BIT);
    GLES20.glEnable(GLES20.GL_DEPTH_TEST);                                  //打开深度检测
    MatrixState.setProjectFrustum(-ratio, ratio, -1, 1, 1, 100); //设置投影矩阵(透视)
    MatrixState.setCamera(cx, 0, cz, 0, 0, 0, 0f, 1.0f, 0.0f);   //设置摄像机观察矩阵
    MatrixState.pushMatrix();                                               //保护现场
    MatrixState.translate(0, -2, 0);                                        //平移变换
    desert.drawSelf(desertId);                                              //绘制表示地面的纹理矩形
        MatrixState.popMatrix();                                            //恢复现场
        GLES20.glEnable(GLES20.GL_BLEND);                                   //开启混合
        GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA,                             //设置混合因子
                GLES20.GL_ONE_MINUS_SRC_ALPHA);
        MatrixState.pushMatrix();                                           //保护现场
        MatrixState.translate(0, -2, 0);                                    //平移变换
        tg.drawSelf(treeId);                                                //绘制树
        MatrixState.popMatrix();                                            //恢复现场
        GLES20.glDisable(GLES20.GL_BLEND);                                  //关闭混合
        GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT);                        //清除深度缓冲
        MatrixState.pushMatrix();                                           //保护现场
        MatrixState.setProjectOrtho(-1f, 1f, -1f, 1f, 1, 100);   //设置投影矩阵(平行)
        MatrixState.setCamera(0, 0, 3, 0f, 0f, 0f, 0f, 1.0f, 0.0f);//设置摄像机观察矩阵
        tr.drawSelf(maskTextureId);                                         //绘制 Alpha 测试用纹理矩形
        MatrixState.popMatrix();                                            //恢复现场
    }
```

• 第2-18行为绘制主场景的代码，与前面章节中绘制物体的方式完全相同。

• 第19-24行为绘制Alpha测试所用纹理矩形的代码，其中首先清除了深度缓冲，然后设置相关矩阵，最后调用纹理矩形的drawSelf方法绘制部分透明的纹理矩形。

> **提示**
>
> 之所以在绘制部分透明的纹理矩形前要清除深度缓冲是因为，此纹理矩形并不属于主场景，不应该与主场景中的物体以同一标准进行深度检测，否则可能会产生不正确的视觉效果。

（2）接着介绍的是实现Alpha测试功能的片元着色器，其代码如下。

代码位置：见随书光盘中源代码/第12章/Sample12_2/assets目录下的frag.sh。

```
precision mediump float;                          //给出默认的浮点精度
varying vec2 vTextureCoord;                       //从顶点着色器传递过来的纹理坐标
uniform sampler2D sTexture;                       //纹理内容数据
void main() {
  vec4 bcolor = texture2D(sTexture, vTextureCoord); //根据纹理坐标采样出颜色值
  if(bcolor.a<0.6) {                              //若Alpha值小于0.6,将片元丢弃
        discard;
  } else {
    gl_FragColor=bcolor;                          //否则将此颜色作为片元的颜色
  }}
```

提示

从上述代码中可以看出，通过片元着色器实现Alpha测试非常简单。仅仅需要将纹理采样所得颜色的Alpha通道值与预定的阈值比较，根据比较的结果决定是否丢弃（discard）片元即可。

本案例的片元着色器中使用了一个以前没有介绍过的操作——discard，此操作的功能为丢弃当前片元。通过此操作在片元着色器中丢弃片元与经过深度、剪裁等测试后丢弃片元是没有本质区别的。只不过前者是渲染管线固定功能部分自动完成的，后者是由开发人员编程完成的，读者可以根据需要选用。

12.3　横板测试

本节将介绍管线固定功能部分所提供的测试中最为灵活的一个——模板测试，介绍时主要包含两方面的内容：基本原理以及一个简单的案例。

12.3.1　基本原理

本章的第一节已经介绍了剪裁测试，通过其应用程序可以很方便地将绘制限定在一个矩形的区域内。但如果需要把绘制限定在一个不规则的区域内，剪裁测试就无能为力了，此时就需要使用模板测试了。模板测试也称为蒙板测试，其用途非常广泛。

例如，需要绘制一个不规则形状的池塘及其周围树木在池塘中倒影的场景，为了保证倒影被正 确绘制在池塘表面而不会越界，可以使用模板测试，具体情况如图12-5所示。

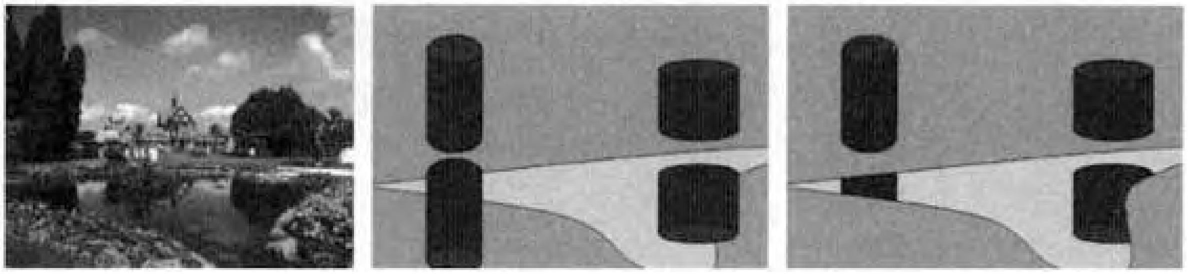


图12-5　模板测试功能示意图

提示

图12-5中左侧为池塘倒影真实场景的照片，中间为不使用模板测试的情况下绘制场景的示意图，右侧为开启模板测试后绘制场景的示意图。从中间与右侧示意图的对比中可以看出，模板测试可以将绘制限制在特定的任意形状区域内以得到正确的视觉效果。

前面第3章在介绍渲染管线时曾经介绍过，帧缓冲中有一个组件——模板缓冲。模板测试就是通过模板缓冲中记录的模板信息来完成的。具体来说就是渲染管线在模板缓冲区中为每个位置的片元保存了一个“模板值”，当像素需要进行模板测试时，将设定的模板参考值与该片元对应位置的模板值进行比较，符合条件的片元通过测试，不符合条件的则被丢弃，不进行渲染。

使用模板测试所需的核心代码如下所列。

```
GLES20.glClear(GLES20.GL_STENCIL_BUFFER_BIT);            //清除模板缓冲
GLES20.glEnable(GLES20.GL_STENCIL_TEST);                 //允许模板测试
GLES20.glStencilFunc(GLES20.GL_ALWAYS, 1, 1);            //设置模板测试参数
GLES20.glStencilOp(                                      //设置模板测试后的操作
    GLES20.GL_KEEP, GLES20.GL_KEEP, GLES20.GL_REPLACE);
GLES20.glDisable(GLES20.GL_STENCIL_TEST);                //禁用模板测试
```

以上代码中，设置模板测试参数的方法glStencilFunc中的第一个参数为比较模式，第二个参数为参考值，第三个参数为掩码（mask）。比较模式包括8种，如表12-1所列。

表12-1　模板测试的8种比较模式

比较模式	含　义
GL_NEVER	从不通过模板测试
GL_ALWAYS	总是通过模板测试
GL_LESS	只有参考值<（模板缓冲区的值&mask）时才通过
GL_LEQUAL	只有参考值<=（模板缓冲区的值&mask）时才通过
GL_EQUAL	只有参考值=（模板缓冲区的值&mask）时才通过
GL_GEQUAL	只有参考值>=（模板缓冲区的值&mask）时才通过
GL_GREATER	只有参考值>（模板缓冲区的值&mask）时才通过
GL_NOTEQUAL	只有参考值!=（模板缓冲区的值&mask）时才通过

> **提示**
>
> 表12-1中的“&”表示的是位与操作，其功能为将模板缓冲区的值与掩码值按位进行与操作。

设置模板测试后操作的方法glStencilOp中3个参数的含义分别如下所列。

- 第一个参数表示模板测试未通过时此片元对应的模板值该如何变化。
- 第二个参数表示模板测试通过，但深度测试未通过时此片元对应的模板值该如何变化。
- 第三个参数表示模板测试和深度测均通过时此片元对应的模板值该如何变化。

> **提示**
>
> 如果没有启用深度测试，则认为深度测试总是通过。

3个参数的可选值情况如表12-2所列。

表12-2　3个参数的可选值

参数值	模板值变化情况
GL_KEEP	不改变
GL_ZERO	回零
GL_REPLACE	使用测试条件中的设定值来代替当前模板值
GL_INCR	增加1，但如果已经是最大值，则保持不变
GL_NCR_WRAP	增加1，但如果已经是最大值，则重新从零开始
GL_DECR	减少1，但如果已经是零，则保持不变
GL_DECR_WRAP	减少1，但如果已经是零，则重新设置为最大值
GL_INVERT	按位取反

提示

假定模板缓冲区中每个像素采用了n个比特，则模板值的取值范围为0~2^n -1。

实际开发中，要根据具体的需求灵活组合使用表12-2中列出的操作。例如前面提到的绘制池塘中倒影的情况应该包括下列操作步骤。

（1）关闭模板测试，绘制出地面和树木。

（2）开启模板测试，使用glClear方法设置所有片元位置的模板值为0。

（3）设置下列参数组合后绘制池塘水面。

```
GLES20.glStencilFunc(GLES20.GL_ALWAYS, 1, 1);          //设置模板测试参数
GLES20.glStencilOp(                                    //设置模板测试后的操作
    GLES20.GL_KEEP, GLES20.GL_KEEP, GLES20.GL_REPLACE);
```

提示

完成了池塘水面的绘制后，模板缓冲中有池塘水面片元的位置对应的模板值为否则为0。

（4）设置下列参数组合后绘制倒影。

```
GLES20.glStencilFunc(GLES20.GL_ EQUAL, 1, 1);                    //设置模板测试参数
GLES20.glStencilOp(                                              //设置模板测试后的操作
    GLES20.GL_KEEP, GLES20.GL_KEEP, GLES20.GL_KEEP);
```

提示

这样一来，只有对应位置模板值为1的片元才会被绘制，也就是只有“水面”中的片元才有可能被倒影的片元替换，而其他片元则保持不变。

12.3.2　一个简单的案例

上一小节介绍了模板测试的基本原理，本小节将给出一个使用了模板测试的案例——Sample12_3，其运行效果如图12-6所示。同时，为了对比没有启用模板测试的情况，图12-7给出了关闭了模板测试后案例Sample12_3运行的情况。

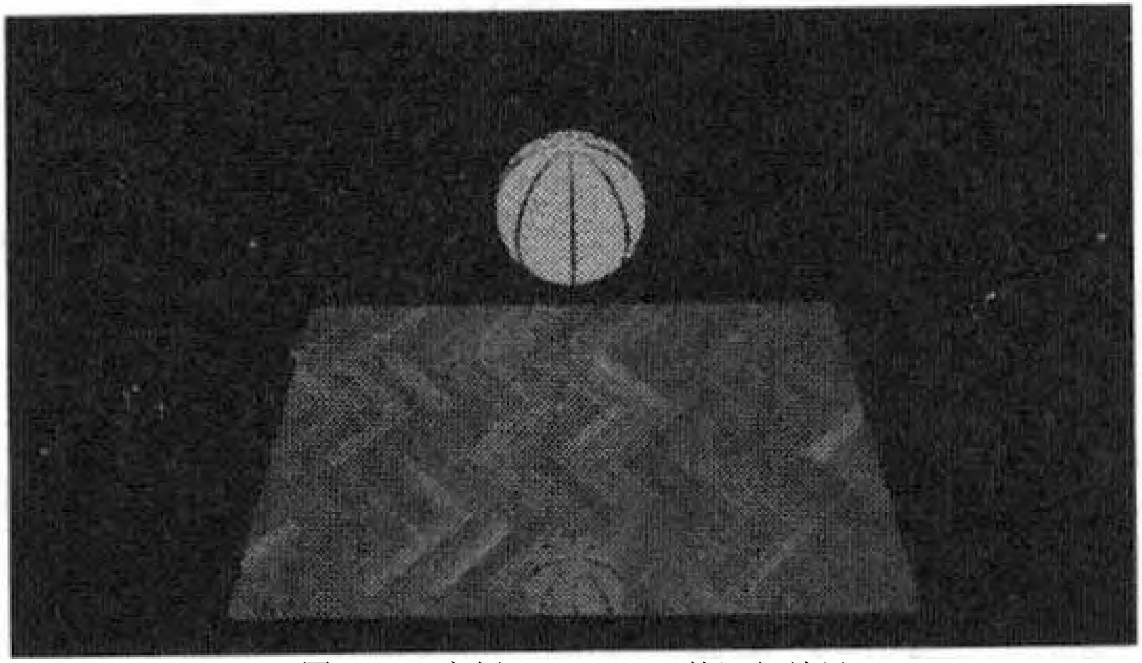


图12-6　案例Sample12_3的运行效果

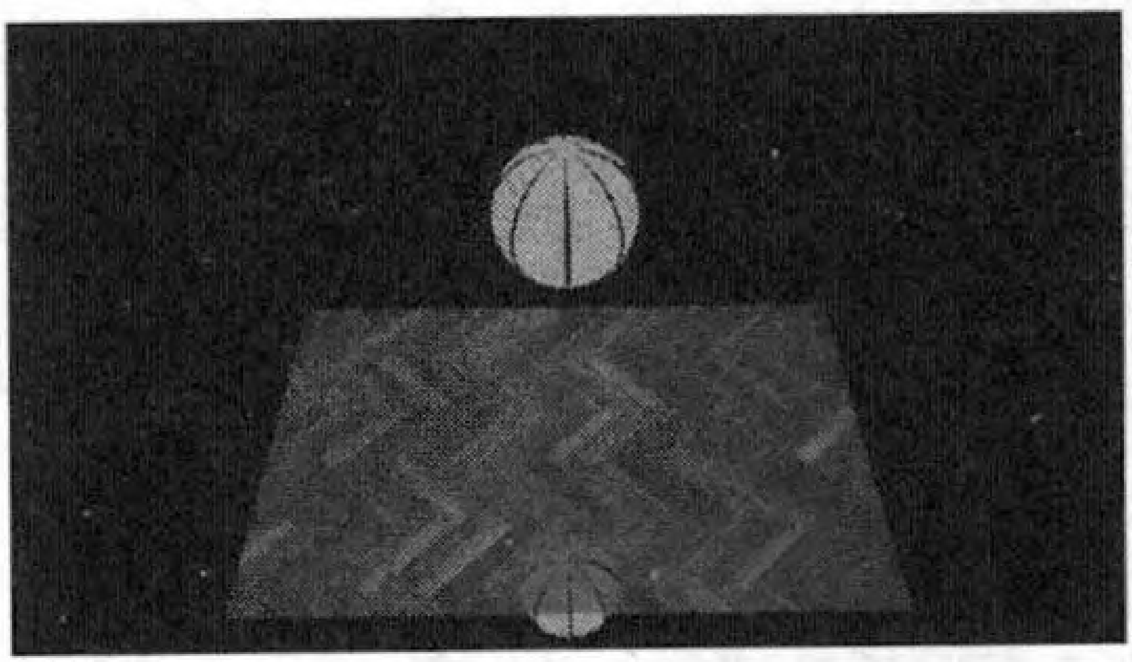


图12-7　禁止模板测试后的运行效果

提示

从图12-6和图12-7的对比中可以看出，此案例中使用模板测试的作用是使篮球的镜像体不会绘制到地板以外的区域。另外，经笔者在采用不同GPU的真机上测试，发现部分采用Adreno系列GPU的机器不能正常渲染出模板测试应有的效果，无论是否开启模板测试渲染效果都等同于关闭模板测试的情况，而采用PowerVR系列GPU的机器都没有问题。

了解了案例的运行效果后，接下来将对本案例的具体开发过程进行介绍。由于本案例主要是对前面第11章案例Sample11_8的升级，因此在这里仅给出本案例中特殊的、有代表性的部分，具体内容如下。

代码位置：见随书光盘中源代码/第12章/Sample12_3/com/bn/Sample12_3目录下的MySurfaceView.java。

```
public void onDrawFrame(GL10 gl) {                          //绘制一帧画面的方法
    GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT               //清除深度缓冲与颜色缓冲
            | GLES20.GL_COLOR_BUFFER_BIT);
    MatrixState.pushMatrix();                               //保护现场
    MatrixState.translate(0, -2, 0);                        平移变换
    GLES20.glClear(GLES20.GL_STENCIL_BUFFER_BIT);           //清除模板缓存
    GLES20.glEnable(GLES20.GL_STENCIL_TEST);                //允许模板测试
    GLES20.glStencilFunc(GLES20.GL_ALWAYS, 1, 1);           //设置模板测试参数
    GLES20.glStencilOp(GLES20.GL_KEEP,                      //设置模板测试后的操作
          GLES20.GL_KEEP, GLES20.GL_REPLACE);
        texRect.drawSelf(textureFloor);                     //绘制反射面地板
        GLES20.glStencilFunc(GLES20.GL_EQUAL, 1, 1);        //设置模板测试参数
        GLES20.glStencilOp(                                 //设置模板测试后的操作
```

```
14            GLES20.GL_KEEP, GLES20.GL_KEEP, GLES20.GL_KEEP);
15        bfd.drawSelfMirror(textureBallId);                    //绘制镜像体
16        GLES20.glDisable(GLES20.GL_STENCIL_TEST);             //禁用模板测试
17        GLES20.glEnable(GLES20.GL_BLEND);                     //开启混合
18        GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA,               //设置混合因子
19                GLES20.GL_ONE_MINUS_SRC_ALPHA);
20        texRect.drawSelf(textureFloorBTM);                    //绘制半透明地板
21        GLES20.glDisable(GLES20.GL_BLEND);                    //关闭混合
22        bfd.drawSelf(textureBallId);                          //绘制实际物体
23        MatrixState.popMatrix();                              //恢复现场
24    }
```

- 上述代码中第6-16行为本案例的关键，这部分的代码主要是实现了上一小节所介绍的绘制步骤，将镜像体的绘制仅限制在特定的区域内。
- 同时从上述代码中可以看出，使用模板测试一般要绘制至少两次，第一次用来产生存储了特殊区域信息模板值的模板缓冲，第二次利用模板缓冲中的值进行测试使得绘制仅被限制在特定的区域内。

12.4　任意剪裁平面

实际开发中偶尔会有这样的需求，仅绘制某一特定平面所确定半空间中的物体部分，其他部分不予绘制，就像物体被切掉了一部分一样。此时就可以使用任意剪裁平面技术，本小节将对其进行详细的介绍，首先会给出任意剪裁平面的基本原理，接下来会给出一个使用任意剪裁平面的案例。

12.4.1　基本原理

除了视景体的6个剪裁平面（左、右、底、顶、近和远）之外，用户还可以再指定其他任意平面进行剪裁。剪裁平面可以用于删除场景中无关的物体部分，例如用于显示物体的剖面视图。OpenGL ES 1.x中提供了最多可达6个任意剪裁平面的支持，使用时只要给出所用剪裁平面的参数即可。

但OpenGL ES 2.0中不直接支持这项功能，需要由开发人员在片元着色器中通过编程实现，下面给出了在OpenGL ES 2.0中实现任意剪裁平面所需的工作。

（1）给出用于定义剪裁平面的4个参数A、B、C、D，这4个参数分别是平面解析方程（$Ax+By+Cz+D=0$）中的4个系数。

（2）将剪裁平面的4个参数传入渲染管线，以备着色器使用。

（3）在顶点着色器中判断顶点是否在平面某的一侧，具体的方法为：将顶点位置（x0,y0,z0）代入平面方程$Ax+By+Cz+D=0$，完成计算后将得到的值传入片元着色器。

（4）片元着色器中根据接收到的$Ax0+By0+Cz0+D$表达式的值与0之间的关系就可以得出顶点与剪裁平面之间的位置关系，以决定是否丢弃片元。

提示

若$Ax0+By0+Cz0+D>0$，则顶点在平面的一侧，反之在平面的另一侧。

12.4.2　茶壶被任意平面剪裁的案例

上一小节介绍了剪裁平面的基本原理，本小节通过一个案例（Sample12_4）来介绍任意剪裁平面的使用，其运行效果如图12-8所示。



图12-8　案例Sample12_4的运行效果图

说明

图12-8从左至右给出了一个茶壶被3个不同参数的剪裁平面剪裁的情况。

了解了案例的运行效果后，接下来将对本案例的具体开发过程进行介绍。由于本案例主要是对前面第9章案例Sample9_3的升级，因此在这里仅给出本案例中特殊的、有代表性的部分，具体内容如下。

（1）首先介绍MySurfaceView类中用于绘制一帧画面的onDrawFrame方法，其代码如下。

代码位置：见随书光盘中源代码/第12章/Sample12_4/com/bn/Sample12_4目录下的MySurfaceView.java。

```
public void onDrawFrame(GL10 gl) {                          //绘制一帧画面的方法
    if (countE >= 2) {                                      //若参考值大于 2 则步进变为-0.01
        spanE = -0.01f;
    } else if (countE <= 0) {                               //若参考值小于 0 则步进变为 0.01
        spanE = 0.01f;
    }
    countE = countE + spanE;                                //改变剪裁平面解析方程中的参数
    float e[] = { 1, countE - 1, -countE + 1, 0 };//定义裁剪平面解析方程中 4 个参数的所属数组
    GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT   //清除深度缓冲与颜色缓冲
                | GLES20.GL_COLOR_BUFFER_BIT);
        MatrixState.pushMatrix();                           //保护现场
        MatrixState.translate(0, -2f, -25f);                //平移变换
        MatrixState.rotate(yAngle, 0, 1, 0);                //绕 Y 轴旋转
        MatrixState.rotate(zAngle, 1, 0, 0);                //绕 Z 轴旋转
        if (lovo != null) { lovo.drawSelf(e); }             //若加载的物体不为空则绘制物体
        MatrixState.popMatrix();                            //恢复现场
}
```

说明

上述代码中主要是增加了在绘制每一帧画面前对平面解析方程中的参数进行微调，并将平面解析方程参数数组传递给绘制方法drawSelf的代码，其他的部分基本没有变化。

（2）接着有变化的就是在物体类LoadedObjectVertexNormal中增加了将平面解析方程参数数组传递进渲染管线的方法，此部分代码与将光源位置传递进渲染管线的代码套路完全相同，这里不再 赘述，需要的读者请自行查阅随书光盘中的源代码。

（3）完成了Java代码的介绍后，下面介绍实现了任意剪裁平面的着色器。首先是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第12章/Sample12_4/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                        //总变换矩阵
uniform mat4 uMMatrix;                          //基本变换矩阵
uniform vec3 uLightLocation;                    //光源位置
uniform vec3 uCamera;                           //摄像机位置
attribute vec3 aPosition;                       //顶点位置
attribute vec3 aNormal;                         //顶点法向量
uniform vec4 u_clipPlane;                       //剪裁平面参数
varying vec4 ambient;                           //用于传递给片元着色器的环境光最终强度
varying vec4 diffuse;                           //用于传递给片元着色器的散射光最终强度
varying vec4 specular;                          //用于传递给片元着色器的镜面光最终强度
varying float u_clipDist;                       //用于传递给片元着色器的剪裁信息
……//此处省略了计算光照的pointLight方法，读者可自行查看随书光盘的源代码
void main() {
  gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点的位置
  pointLight(normalize(aNormal),ambient,diffuse,specular,uLightLocation, //计算光照
    vec4(0.1,0.1,0.1,1.0),vec4(0.7,0.7,0.7,1.0),vec4(0.3,0.3,0.3,1.0));
  //将顶点位置代入平面方程Ax+By+Cz+D，并将计算结果传递给片元着色器
  u_clipDist = dot(aPosition.xyz, u_clipPlane.xyz) +u_clipPlane.w;
}
```

提示

上述顶点着色器的代码中主要是增加了将顶点位置代入平面解析方程，并将计算结果通过易变变量传递给片元着色器的代码。

（4）介绍完顶点着色器后，接下来介绍片元着色器，其代码如下。

代码位置：见随书光盘中源代码/第12章/Sample12_4/assets目录下的frag.sh。

```
1 precision mediump float;                                  //给出默认的浮点精度
2 varying vec4 ambient;                                     //从顶点着色器传递过来的环境光最终强度
3 varying vec4 diffuse;                                     //从顶点着色器传递过来的散射光最终强度
4 varying vec4 specular;                                    //从顶点着色器传递过来的镜面光最终强度
5 varying float u_clipDist;                                 //从顶点着色器传递过来的剪裁信息
6 void main() {
7    if(u_clipDist < 0.0) discard;                          //如果剪裁信息小于0，则将片元丢弃
8    vec4 finalColor=vec4(0.76471,0.51765,0.44705,1.0);     //物体的颜色
9    //综合3个通道光的最终强度及片元的颜色计算出最终片元的颜色并传递给管线
10       gl_FragColor = finalColor*ambient+finalColor*specular+finalColor*diffuse;
11   }
```

提示

从上述片元着色器的代码中可以看出，最主要的就是第7行增加的根据接收的剪裁信息值是否小于0来决定是否丢弃片元的代码，其他的部分基本没有变化。

12.5　本章小结

本章主要介绍了几种常用的测试与任意剪裁平面，包括剪裁测试、Alpha测试、模板测试及任意剪裁平面。通过本章的学习，读者可以在实际开发中利用这些知识渲染出更加酷炫的3D场景。

第13章

顶点着色器的妙用

前面的章节中已经大量地用到了顶点着色器，但并没有能完全体现出顶点着色器的功效。本章将进一步介绍一些顶点着色器的使用技巧，通过应用这些技巧可以开发出很多酷炫的效果，而这些效果在OpenGL ES 1.x时代是很难高效地实现的。

13.1　飘扬的旗帜

飘扬的旗帜是本章要介绍的第一个案例，通过此技术可以实现类似旗帜迎风飘扬的效果，也可以实现类似水面起伏的效果。

13.1.1　基本原理

介绍本案例的具体开发步骤之前首先需要了解一下实现旗帜飘扬的基本原理，如图13-1所示。

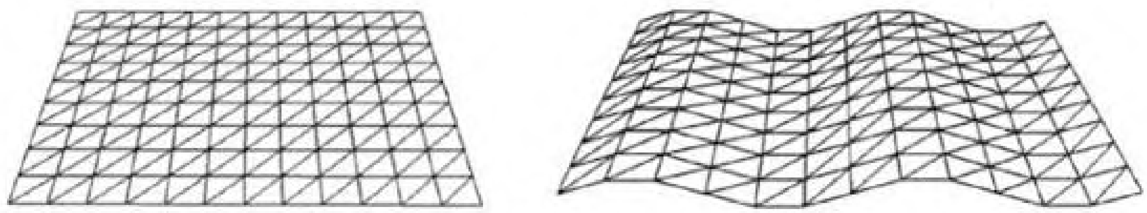


图13-1　旗帜飘扬的线框图

提示

图13-1中左图为原始情况下旗帜的顶点位置情况，右图为顶点着色器根据参数计算后某一帧画面中旗帜的顶点位置情况。

从图13-1中可以看出，矩形的旗帜与以前案例中绘制的矩形不同，不再是仅有两个三角形组成的整体，而是有大量的小三角形组成的。这样只要在绘制一帧画面时由顶点着色器根据一定的规则变换各个顶点的位置，即可得到旗帜迎风飘动的效果。

为了使旗帜的飘动过程比较平滑，本案例采用的是基于正弦曲线的顶点位置变换规则，具体情况如图13-2所示。

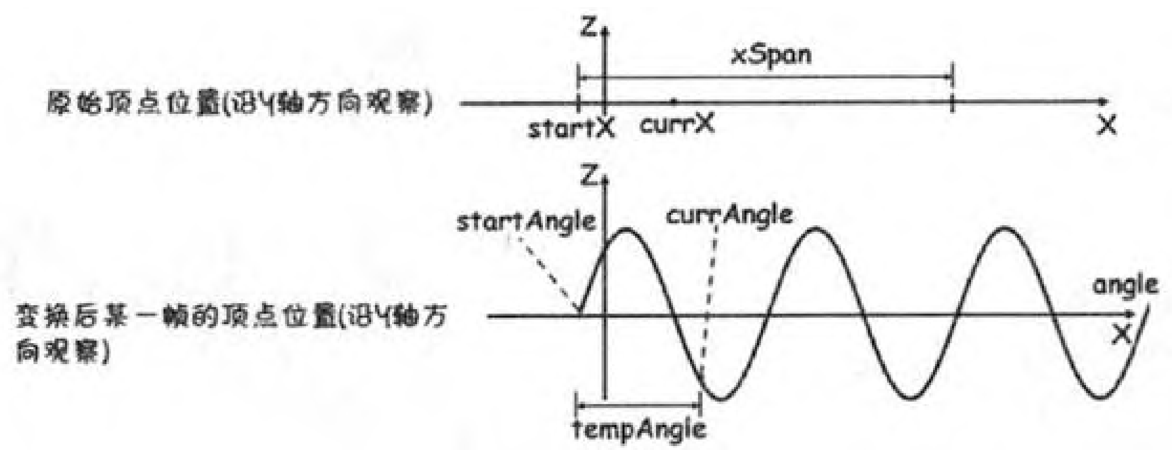


图13-2　X方向波浪原理图

说明

图13-2给出的是旗帜面向Z轴正方向，即顶点沿Z轴上下振动，形成的波浪沿*X*轴传播的情况。同时注意，观察的方向是沿*Y*轴的方向。

从图13-2中可以看出，传入顶点着色器的原始顶点的*Z*坐标都是相同的（本案例中为0），经过顶点着色器变换后顶点的*Z*坐标是根据正弦曲线分布的。具体的计算方法如下。

- 首先计算出当前处理顶点的*X*坐标与最左侧顶点*X*坐标的差值，即X距离。
- 然后根据距离与角度的换算率将X距离换算为当前顶点与最左侧顶点的角度差（tempAngle）。

提示

所谓距离与角度的换算率指的是由开发人员人为设定的一个值，将距离乘以其后就可以换算成角度值。例如可以规定，X方向上距离4等于2π，则换算公式为：X距离×2π/4。

- 接着将tempAngle加上最左侧顶点的对应角度（startAngle）即可得到当前顶点的对应角度（currAngle）。
- 最后通过求currAngle的正弦值即可得到当前顶点变换后的*Z*坐标。

可以想象出，只要在绘制每帧画面时传入不同的startAngle值（例如在0～2π连续变化），即可得到平滑的基于正弦曲线的旗帜飘扬的动画了。

13.1.2 开发步骤

上一小节介绍了飘扬旗帜的基本原理，本小节将基于此原理开发一个旗帜迎风飘扬的案例Sample13_1，其运行效果如图13-3所示。



图13-3　案例Sample13_1的运行效果图

提示

图13-3中从左到右分别为x方向波浪、斜向下方向波浪和xy双向波浪的效果。由于插图是灰度印刷且是静态的，因此可能看得不是很清楚，建议读者用真机运行本案例进行体会。

了解了案例的运行效果后，接下来对本案例的具体开发过程进行简要介绍。由于本案例中的大部分类和前面章节很多案例中的非常类似，因此在这里只给出本案例中比较有代表性的部分，具体内容如下所列。

（1）首先需要简单说明的是，表示旗帜的纹理矩形类TextureRect，其大部分代码与本书前面的很多案例基本一致。主要的区别是需要增加将起始角度和角度总跨度等数据传入渲染管线的相关代码，有了这些数据后顶点着色器在绘制每帧画面前就可以顺利地对顶点位置进行变换了。

提示

增加的代码非常简单，需要的读者请自行查阅随书光盘。

（2）从案例效果图中已经可以看出，本案例中的波浪方向有3种选择，因此需要3套着色器来实现不同的波浪方向。首先给出最简单的实现X方向波浪的顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第13章/Sample13_1/assets目录下的vertex_tex_x.sh。

```
1 uniform mat4 uMVPMatrix;                          //总变换矩阵
2 uniform float uStartAngle;                        //本帧起始角度(即最左侧顶点的对应角度)
3 uniform float uWidthSpan;                         //横向长度总跨度
4 attribute vec3 aPosition;                         //顶点位置
5 attribute vec2 aTexCoor;                          //顶点纹理坐标
6 varying vec2 vTextureCoord;                       //用于传递给片元着色器的纹理坐标
7 void main(){
8    float angleSpanH=4.0*3.14159265;          //横向角度总跨度，用于进行X距离与角度的换算
9    float startX=-uWidthSpan/2.0;             //起始X坐标(即最左侧顶点的X坐标)
10       //根据横向角度总跨度、横向长度总跨度及当前点X坐标折算出当前顶点X坐标对应的角度
11       float currAngle=uStartAngle+((aPosition.x-startX)/uWidthSpan)*angleSpanH;
12       float tz=sin(currAngle)*0.1;          //通过正弦函数求出当前点的Z坐标
13       //根据总变换矩阵计算此次绘制此顶点位置
14       gl_Position = uMVPMatrix * vec4(aPosition.x,aPosition.y,tz,1);
15       vTextureCoord = aTexCoor;             //将接收的纹理坐标传递给片元着色器
16   }
```

说明

上述顶点着色器实现了上一小节中所介绍的基于正弦曲线的X方向波浪，其中第8行的变量angleSpanH可以用来控制波浪的密度，其值越大，波浪密度越大。

（3）接着给出的是实现斜向下方向波浪的顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第13章/Sample13_1/assets目录下的vertex_tex_xie.sh。

```
uniform mat4 uMVPMatrix;                     //总变换矩阵
uniform float uStartAngle;                   //本帧起始角度(即最左侧顶点的对应角度)
uniform float uWidthSpan;                    //横向长度总跨度
attribute vec3 aPosition;                    //顶点位置
attribute vec2 aTexCoor;                     //顶点纹理坐标
varying vec2 vTextureCoord;                  //用于传递给片元着色器的纹理坐标
void main(){
  float angleSpanH=4.0*3.14159265;           //横向角度总跨度，用于进行X距离与角度的换算
  float startX=-uWidthSpan/2.0;              //起始X坐标(即最左侧顶点的X坐标)
     //根据横向角度总跨度、横向长度总跨度及当前点X坐标折算出当前顶点X坐标对应的角度
     float currAngleH=uStartAngle+((aPosition.x-startX)/uWidthSpan)*angleSpanH;
     float angleSpanZ=4.0*3.14159265;        //纵向角度总跨度，用于进行Y距离与角度的换算
     float uHeightSpan=0.75*uWidthSpan;      //纵向长度总跨度
     float startY=-uHeightSpan/2.0;          //起始Y坐标(即最上侧顶点的Y坐标)
     //根据纵向角度总跨度、纵向长度总跨度及当前点Y坐标折算出当前顶点Y坐标对应的角度
     float currAngleZ=((aPosition.y-startY)/uHeightSpan)*angleSpanZ;
     float tzH=sin(currAngleH-currAngleZ)*0.1;     //通过正弦函数求出当前点的Z坐标
     //根据总变换矩阵计算此次绘制此顶点的位置
     gl_Position = uMVPMatrix * vec4(aPosition.x,aPosition.y,tzH,1);
     vTextureCoord = aTexCoor;               //将接收的纹理坐标传递给片元着色器
  }
```

说明

本质上讲，上述斜向下方向波浪的顶点着色器与前面的X方向波浪的顶点着色器没有本质区别，仅仅是在计算当前顶点的对应角度

时增加了Y轴方向的计算，不再是仅考虑X轴的坐标。因此，形成的波浪方向就是斜向下的。

（4）最后给出的是沿X、Y两个方向各自传播的波浪效果叠加的顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第13章/Sample13_1/assets目录下的vertex_tex_xy.sh。

```
uniform mat4 uMVPMatrix;                          //总变换矩阵
uniform float uStartAngle;                        //本帧起始角度(X、Y两个方向都是其)
uniform float uWidthSpan;                         //横向长度总跨度
attribute vec3 aPosition;                         //顶点位置
attribute vec2 aTexCoor;                          //顶点纹理坐标
varying vec2 vTextureCoord;                       //用于传递给片元着色器的纹理坐标
void main(){
  //首先计算当前顶点X方向波浪对应的Z坐标
  float angleSpanH=4.0*3.14159265;           //横向角度总跨度，用于进行X距离与角度的换算
      float startX=-uWidthSpan/2.0;          //起始X坐标(即最左侧顶点的X坐标)
      //根据横向角度总跨度、横向长度总跨度及当前点X坐标折算出当前顶点X坐标对应的角度
      float currAngleH=uStartAngle+((aPosition.x-startX)/uWidthSpan)*angleSpanH;
      float tzH=sin(currAngleH)*0.1;         //X方向波浪对应的Z坐标
      //接着计算当前顶点Y方向波浪对应的Z坐标
      float angleSpanZ=4.0*3.14159265;       //纵向角度总跨度，用于进行Y距离与角度的换算
      float uHeightSpan=0.75*uWidthSpan;     //纵向长度总跨度
      float startY=-uHeightSpan/2.0;         //起始Y坐标(即最上侧顶点的Y坐标)
      //根据纵向角度总跨度、纵向长度总跨度及当前点Y坐标折算出当前顶点Y坐标对应的角度
      float currAngleZ=uStartAngle+3.14159265/3.0+((aPosition.y-startY)/uHeightSpan)
      *angleSpanZ;
      float tzZ=sin(currAngleZ)*0.1;         //Y方向波浪对应的Z坐标
      //根据总变换矩阵计算此次绘制此顶点的位置,
      gl_Position = uMVPMatrix * vec4(aPosition.x,aPosition.y,tzH+tzZ,1);
      vTextureCoord = aTexCoor;              //将接收的纹理坐标传递给片元着色器
  }
```

提示

本质上讲上述X、Y双向波浪的顶点着色器与前面的X方向波浪的顶点着色器没有本质区别，仅仅是首先分别计算了X方向和Y方向波浪在当前顶点位置的*Z*坐标，最后将两个*Z*坐标叠加实现了波的叠加。因此，运行案例时看到的波浪就是X、Y两个方向的了。

本案例3套着色器中的片元着色器都是一样的，并且都是采用的普通的纹理采样片元着色器，前面很多案例中已经出现过，因此这里不再赘述。

13.2　扭动的软糖

上一节介绍的飘扬的旗帜是对一个纹理矩形中的顶点位置进行了变换，本节的案例将对一个长方体中的顶点位置进行变换以实现软糖扭动的效果。

13.2.1　基本原理

介绍本案例的具体开发之前首先需要了解一下实现的基本原理，具体情况如图13-4所示。

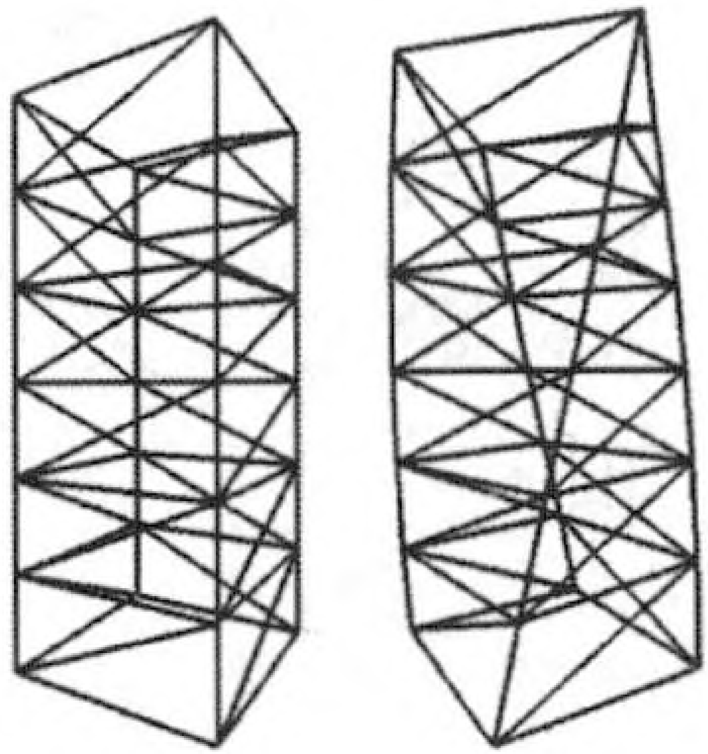


图13-4　软糖的线框图

从图13-4中可以看出，软糖模型实际上是由很多层的小矩形叠加而成。在同一帧中，随着Y坐标的不断升高此层的顶点绕中心轴扭曲的角度越大。因此，实现扭动软糖的效果只要将代表软糖的长方体中各层顶点的X、Z坐标按照一定的规则根据顶点的Y坐标以及当前帧的控制参数进行变换即可，具体的计算思路如图13-5、图13-6与图13-7所示。

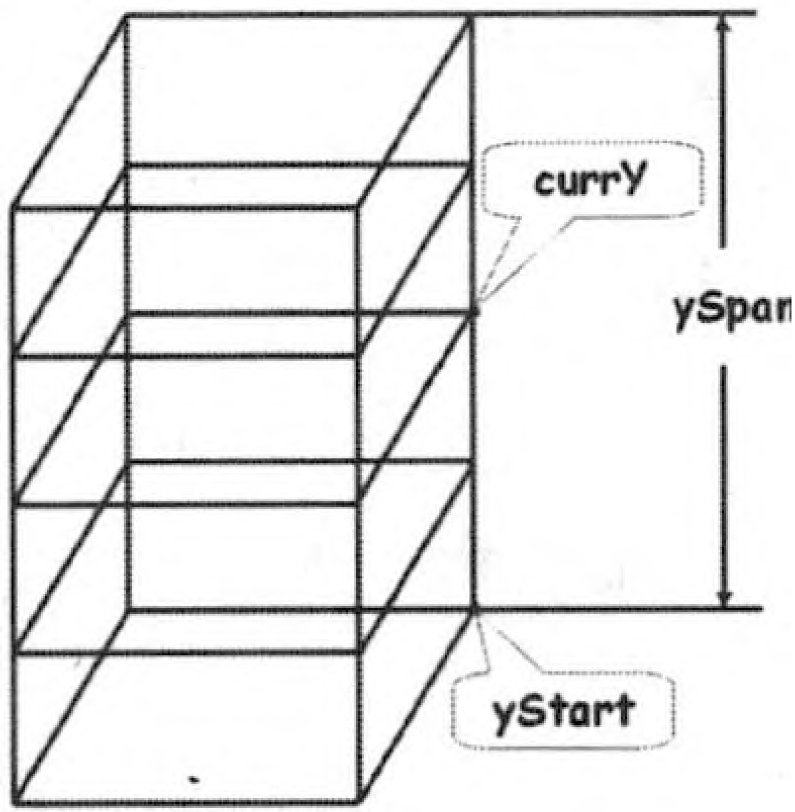


图13-5　软糖扭曲原理图1

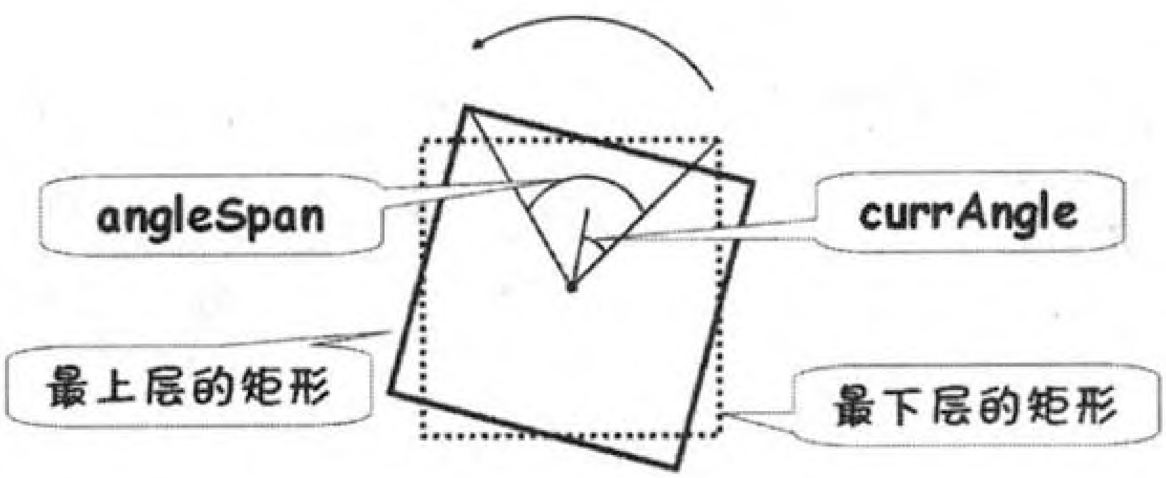


图13-6　软糖扭曲原理图2

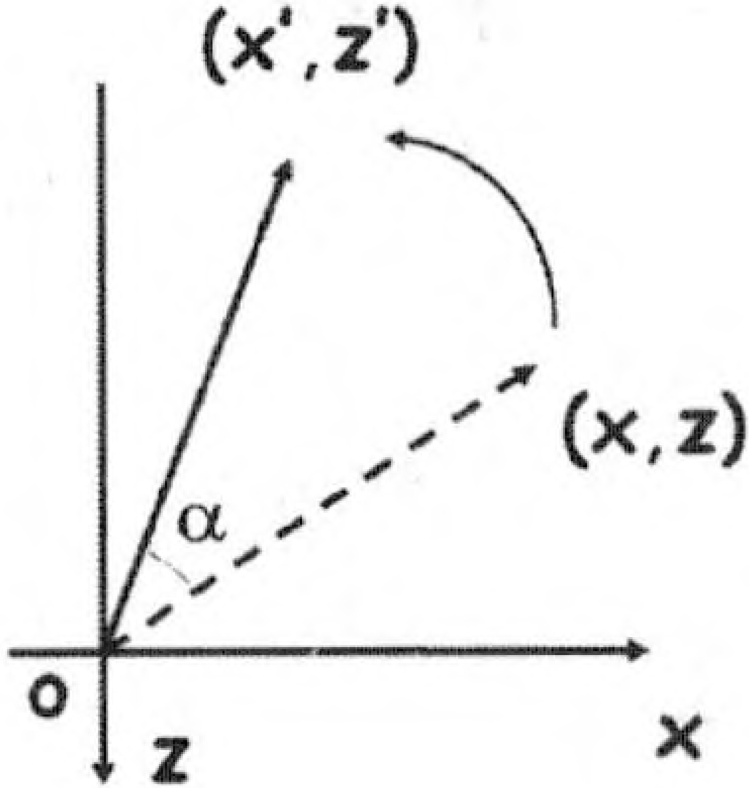


图13-7　向量旋转图

具体的计算步骤如下。

- 首先如图13-5所示，需要计算出当前顶点*Y*坐标与最下层顶点*Y*坐标的差值。
- 接着根据*Y*坐标的差值，角度换算比例以及本帧的总扭曲角度换算出当前顶点的扭曲角度，计算公式为：currAngle =（currY-startY）/ySpan×angleSpan。
- 最后根据当前顶点的*X*、*Z*坐标，扭曲的角度计算出变换后顶点的*X*、*Z*坐标，此步的计算思路如图13-6与图13-7所示。

从图13-6与图13-7中可以看出，将顶点绕中心轴扭曲（旋转）实际上可以看作，将从中心点出发到变换前顶点的向量旋转指定的角度。旋转后得到的新向量的终点位置即所求的变换后顶点的位置，因此具体的计算公式如下。

$x' = x\cos\alpha - z\sin\alpha \qquad z' = x\sin\alpha + z\cos\alpha$

说明

上述公式中的α为需要旋转（扭曲）的角度，实际计算时采用前面步骤计算出来的变量currAngle的值即可。

13.2.2 开发步骤

上一小节介绍了软糖扭动的基本原理，本小节将基于此原理开发一个软糖不断扭动的案例Sample13_2，其运行效果如图13-8所示。

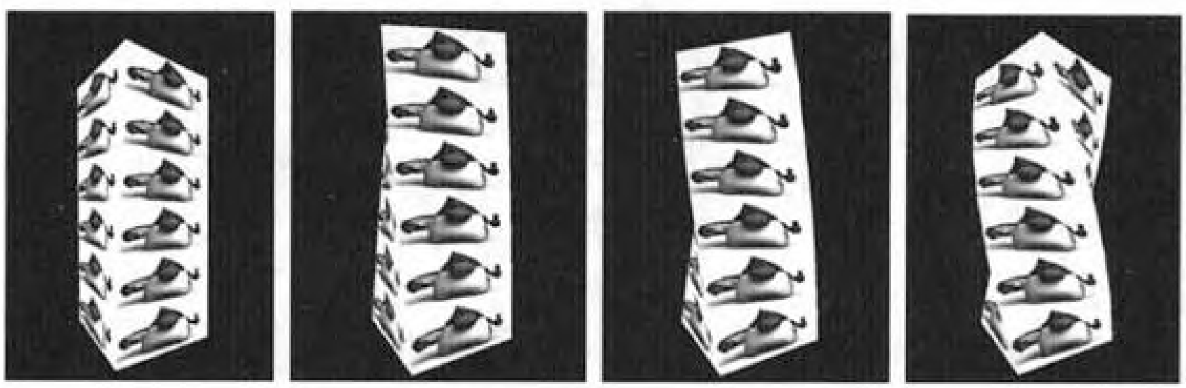


图13-8 案例Sample13_2的运行效果图

说明

图13-8给出了4幅软糖扭动过程中的程序抓图，从左至右扭动角度不断增大。

了解了案例的运行效果后，接下来对本案例的具体开发过程进行简要介绍。由于本案例中的大部分类和前面章节很多案例中的非常类似，因此在这里只给出本案例中比较有代表性的部分，具体内容如下所列。

（1）首先需要简单说明的是，表示软糖的长方体类Cuboid，其大部分代码与本书前面的很多案例基本一致。主要的区别是需要增加将当前帧最大扭动角度、Y起始坐标、*Y* 坐标总跨度等数据传入渲染管线的相关代码，有了这些数据后顶点着色器在绘制每帧画面前就可以顺利地对顶点位置进行变换了。

提示

增加的代码非常简单，需要的读者请自行查阅随书光盘。

（2）接着需要给出的是实现软糖扭动的顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第13章/Sample13_2/assets目录下的vertex_tex.sh。

```
uniform mat4 uMVPMatrix;                                   //总变换矩阵
attribute vec3 aPosition;                                  //顶点位置
attribute vec2 aTexCoor;                                   //顶点纹理坐标
varying vec2 vTextureCoord;                                //用于传递给片元着色器的纹理坐标
uniform float angleSpan;                                   //本帧扭曲总角度
uniform float yStart;                                      //Y坐标起始点
uniform float ySpan;                                       //Y坐标总跨度
void main(){
float tempAS= angleSpan*(aPosition.y-yStart)/ySpan; //计算当前顶点扭动(绕中心点选择)的角度
     vec3 tPosition=aPosition;
     if(aPosition.y>yStart){              //若不是最下面一层的顶点则计算扭动后的X、Z坐标
       tPosition.x=(cos(tempAS)*aPosition.x-sin(tempAS)*aPosition.z);
       tPosition.z=(sin(tempAS)*aPosition.x+cos(tempAS)*aPosition.z);
     }
     gl_Position = uMVPMatrix * vec4(tPosition,1); //根据总变换矩阵计算此次绘制此顶点的位置
     vTextureCoord = aTexCoor;                   //将接收的纹理坐标传递给片元着色器
 }
```

说明

上述顶点着色器根据上一小节介绍的计算过程实现了对顶点位置的变换，其中最核心的就是第12-13行根据扭动角度计算变换后顶点X、Z坐标的代码。

13.3 风吹椰林场景的开发

前两节分别给出了两个单一的用顶点着色器实现软体的案例，本节将给出一个综合性的软体案例。此案例为风吹海滩上椰子林的场景，场景中海浪拍打沙滩，椰子树在风的吹动下摇摆，伴随着海浪的声音，非常吸引人。

提示

本案例中的海浪实际就是放平的、采用了海水纹理的飘扬的旗帜，天空采用的是天空穹，海岛采用的是灰度图地形，海浪的声音采用的是声音池。这些技术在前面的章节中都已经详细介绍过，因此本节就不再赘述。而椰子树随风摆动是本案例的重点，下面将详细进行介绍。

13.3.1 椰子树随风摇摆的基本原理

介绍椰子树的具体开发之前首先需要了解一下沙滩椰子树随风摆动的基本原理。本案例中椰子树的树干会随着风力的大小、方向产生对应的弯曲，下面的图13-9给出了如何计算某一帧中树干上指定顶点弯曲后位置的策略。

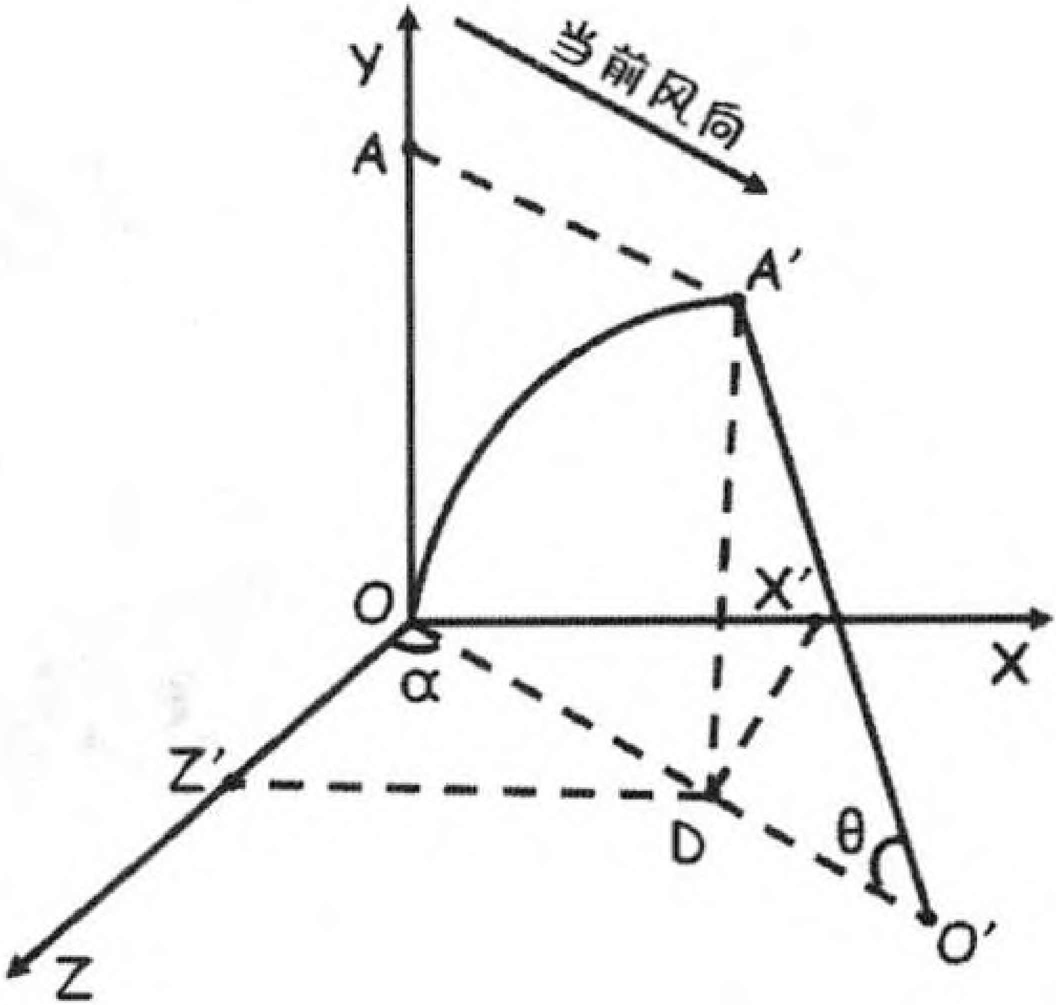


图13-9　椰子树树干弯曲原理图

从图13-9中可以看出，为了简化计算，本案例中采用的风向是与XOZ平面平行的。设当前风向与*Z* 轴正方向的夹角为α，树干原始状态下与*Y* 轴重合。点A为树干模型中的任一顶点，在风的吹动下偏转到A′点。

则顶点着色器需要计算的问题为：已知A点坐标（X0，Y0，Z0）、当前风向与*Z* 轴正方向的夹角α以及弧OA′所在圆的半径OO′，求A点偏转到A′点后的坐标。

提示

本案例采用的计算模型中，半径OO′的大小与风力的大小是成反比例的，风力越大，半径OO′越小。这样就非常容易地实现了风越大，树干弯曲得越厉害。

下面给出了具体的计算步骤。

（1）由于OA′为半径为OO′的一段圆弧，那么可以得出OA′=OA， 且O′O=O′A′ 。

（2）根据弧长公式，可得出树干弯曲后的弧对应的圆心角θ的弧度计算公式如下。

θ = OA′/OO′=OA/OO′

（3）从图13-9以及根据三角函数的知识可以得出如下结论。

A′D=O′A′×sin（θ）=O′O′×sin（OA/OO′）

OD=OO′-O′A′×cos（θ）= OO′-O′O′×cos（OA/OO′）

（4）接着可以得出如下结论。

OX′=OD×sin（α）=（OO′-O′O′×cos（OA/OO′））×sin（α）

OZ′=OD×cos（α）=（OO′-O′O′×cos（OA/OO′））×cos（α）

（5）设顶点A的坐标为（X0，Y0，Z0），偏移后A′的坐标为（X1，Y1，Z1）。则可以用Y0替换上面的OA，那么有如下结论。

OX′=（OO′-OO′×cos（Y0/OO′））×sin（α）

OZ′=（OO′-OO′×cos（Y0/OO′））×cos（α）

（6）最后可以得到A′点的坐标为。

X1=X0+OX′=X0+（OO′-OO′×cos（Y0/OO′））× sin（α）

Y1=A′D=OO′×sin（Y0/OO′）

Z1=Z0+OZ′=Z0+（OO’-OO′×cos（Y0/OO’））× cos（α）

从上述得出的顶点位置变换公式中可以看出，只需要改变风向角度α，就可以使椰子树向不同的方向摆动。同时，只需要根据风力大小改变弯曲半径OO′的大小，就可以改变椰子树树干的弯曲程度。

13.3.2 开发步骤

上一小节介绍了树干弯曲的基本原理，本小节将基于此原理开发一个呈现风吹椰林场景的案例Sample13_3，其运行效果如图13-10所示。

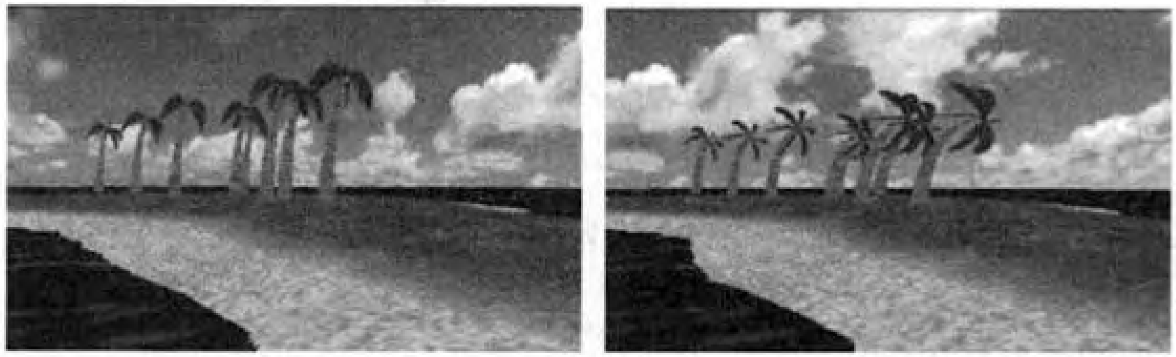


图13-10　案例Sample13_3的运行效果图

本案例运行时可以通过手指在屏幕上左右滑动使摄像机绕场景转动，上下滑动使摄像机推近或远离场景。通过单击手机上的菜单键，程序会弹出菜单。选择菜单中的风向选项可以设置风向，选择菜单中的风力选项可以设置风力，如图13-11所示。

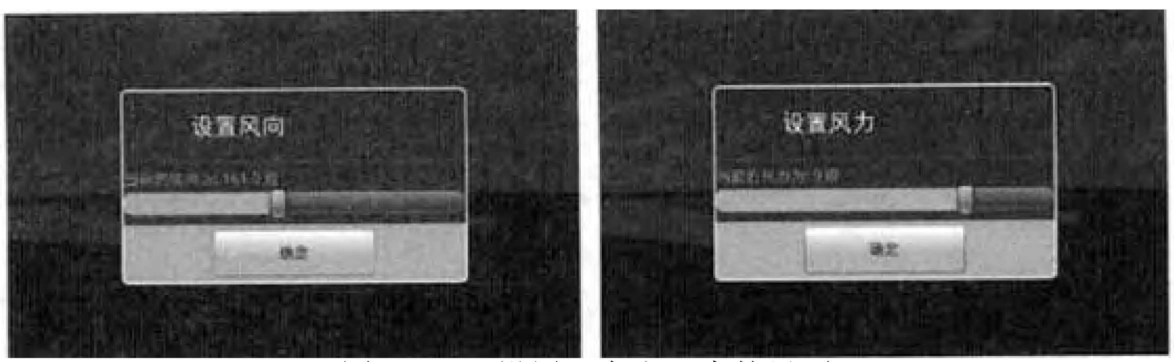


图13-11　设置风向和风力的界面

了解了案例的运行效果后，接下来将对本案例的具体开发过程进行简要介绍。由于本案例中的大部分类和前面章节很多案例中的非常类似，因此，在这里只给出本案例中比较有代表性的与椰子树相关的部分，具体内容如下所列。

（1）首先给出的是用于生成椰子树树干原始位置顶点坐标的initVertexData方法，其来自于表示椰子树树干的TreeTrunk类，具体代码如下。

代码位置：见随书光盘中源代码/第13章/Sample13_3/com/bn/Sample13_3目录下的TreeTrunk.java。

```
public void initVertexData(float bottom_radius,float joint_Height,int jointNum,int availableNum){
  List<Float> vertex_List=new ArrayList<Float>();                //顶点坐标列表
  List<float[]> texture_List=new ArrayList<float[]>();           //顶点纹理坐标列表
  for(int num=0;num<availableNum;num++){      //循环计算出每节树干中的各个顶点
     float temp_bottom_radius=bottom_radius*(jointNum-num)/(float)jointNum;
                                                                 //此节树干底端半径
     float temp_top_radius=bottom_radius*(jointNum-(num+1))/(float)jointNum;
                                                                 //此节树干顶端半径
     float temp_bottom_height=num*joint_Height;                  //此节树干底端的 Y 坐标
     float temp_top_height=(num+1)*joint_Height;                 //此节树干顶端的 Y 坐标
     //循环一周，生成组成此节树干各个四边形的顶点坐标，并卷绕成三角形
        for(float hAngle=0;hAngle<360;hAngle=hAngle+longitude_span) {
            //当前四边形左上点 X、Y、Z 坐标
            float x0=(float) (temp_top_radius*Math.cos(Math.toRadians(hAngle)));
            float y0=temp_top_height;
            float z0=(float) (temp_top_radius*Math.sin(Math.toRadians(hAngle)));
            //当前四边形左下点 X、Y、Z 坐标
            float x1=(float) (temp_bottom_radius*Math.cos(Math.toRadians(hAngle)));
            float y1=temp_bottom_height;
            float z1=(float) (temp_bottom_radius*Math.sin(Math.toRadians(hAngle)));
            //当前四边形右上点 X、Y、Z 坐标
            float x2=(float) (temp_top_radius*Math.cos(Math.toRadians(hAngle+
            longitude_span)));
            float y2=temp_top_height;
            float z2=(float) (temp_top_radius*Math.sin(Math.toRadians(hAngle+
            longitude_span)));
            //当前四边形右下点 X、Y、Z 坐标
            float x3=(float) (temp_bottom_radius*Math.cos(Math.toRadians(hAngle+
            longitude_span)));
            float y3=temp_bottom_height;
            float z3=(float) (temp_bottom_radius*Math.sin(Math.toRadians(hAngle+
            longitude_span)));
            //将顶点坐标按照卷绕成两个三角形的顺序依次放入顶点坐标列表
            vertex_List.add(x0);vertex_List.add(y0);vertex_List.add(z0);
            vertex_List.add(x1);vertex_List.add(y1);vertex_List.add(z1);
            vertex_List.add(x2);vertex_List.add(y2);vertex_List.add(z2);
```

```
            vertex_List.add(x2);vertex_List.add(y2);vertex_List.add(z2);
             vertex_List.add(x1);vertex_List.add(y1);vertex_List.add(z1);
             vertex_List.add(x3);vertex_List.add(y3);vertex_List.add(z3);
        }
        ……//此处省略了计算纹理坐标以及将顶点坐标与纹理坐标送入缓冲的代码
    }
```

提示

从上述代码中可以看出，椰子树的树干是由一节一节的圆形梯台组合而成的。每一节都是下面的半径大，上面的半径小，这也符合现实世界椰子树树干的情况。

（2）为了使树干能够根据风向与风力摆动，还需要在TreeTrunk类中增加将当前风向以及风力对应的树干曲率半径数据传入渲染管线的相关代码。由于将这两项数据传入渲染管线的代码与传递其他数据的代码没有本质区别，故这里不再赘述，需要的读者可以自行查看随书光盘中的源代码。

（3）接着给出的是根据风力、风向对树干顶点位置进行变换的顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第13章/Sample13_3/assets目录下的vertex_tree.sh。

```
uniform mat4 uMVPMatrix;                              //总变换矩阵
uniform float bend_R;                                 //这里指的是树干弯曲的半径
uniform float direction_degree;                       //用角度表示的风向，沿 Z 轴正方向逆时针旋转
attribute vec3 aPosition;                             //顶点位置
attribute vec2 aTexCoor;                              //顶点纹理坐标
varying vec2 vTextureCoord;                           //用于传递给片元着色器的纹理坐标
void main(){
   float curr_radian=aPosition.y/bend_R;              //计算当前的弧度
   float result_height=bend_R*sin(curr_radian);       //计算当前点变换后的 Y 坐标
      float increase=bend_R-bend_R*cos(curr_radian);
      float result_X=aPosition.x+increase*sin(radians(direction_degree));
                                                      //计算当前点最后的 x 坐标
      float result_Z=aPosition.z+increase*cos(radians(direction_degree));
                                                      //计算当前点最后的 z 坐标
      vec4 result_point=vec4(result_X,result_height,result_Z,1.0); //最后结果顶点的坐标
      gl_Position = uMVPMatrix * result_point;//根据总变换矩阵计算此次绘制此顶点的位置
      vTextureCoord = aTexCoor;                        //将接收的纹理坐标传递给片元着色器
   }
```

说明

上述顶点着色器实现了上一小节介绍的顶点随风力、风向变换的算法。读者要想彻底掌握最好比对上一小节介绍的原理研读代码，直接看代码可能难于理解。

（4）介绍完树干部分后，下面介绍一下树叶随风摆动的相关代码。本案例中的树叶采用的是纹理矩形实现的，每棵椰子树有6片树叶（6个纹理矩形）。树叶会根据风向、风力改变位置姿态，本身不会发

生形变。首先给出用于绘制树叶的纹理矩形的顶点及纹理坐标生成方法initVertexData，其来自于TreeLeaves类，具体代码如下。

代码位置：见随书光盘中源代码/第13章/Sample13_3/com/bn/Sample13_3目录下的TreeLeaves.java。

```
public void initVertexData(float width,float height,float absolute_height,int index) {
  vCount=6;
  float vertices[]=null;                                        //顶点坐标数组
  float texCoor[]=null;                                         //纹理坐标数组
  switch(index) {    //根据情况编号生成对应角度树叶纹理矩形的顶点数据
    case 0:          //第一种情况，树叶纹理矩形的边与X轴重合，对应旋转角度为0°
      vertices=new float[]{  0,height+absolute_height,0, 0,absolute_height,0,
        width,height+absolute_height,0, width,height+absolute_height,0,
           0,absolute_height,0,  width,absolute_height,0,
          };
          texCoor=new float[]{ 1,0, 1,1, 0,0,  0,0, 1,1, 0,1 }; //纹理坐标
          terX=width/2;  enterZ=0;                               //确定中心点坐标
        break;
        case 1:         //第一种情况，与X轴夹角60°的树叶纹理矩形
        ……//此处省略了后面5种不同情况的代码，与第一种情况套路完全相同
    }
```

提示

上述initVertexData方法的主要功能为根据情况编号生成对应角度树叶纹理矩形的顶点数据，情况编号从0~5，分别对应0°、60°、120°、180°、240°、300°等6种情况。

（5）接下来给出的是根据当前帧对应的风向、风力计算出树叶纹理矩形位置与姿态数据的resultPoint方法，其来自TreeLeavesControl类，具体代码如下。

代码位置：见随书光盘中源代码/第13章/Sample13_3/com/bn/Sample13_3目录下的TreeLeavesControl.java。

```
1 public float[] resultPoint(float direction_degree,float currBend_R,float pointX,float pointY,float pointZ){
2 float []position=new float[6];                                   //记录位置、姿态数据的数组
3 float curr_radian=pointY/currBend_R;                             //计算当前的弧度
4 float result_Y=(float) (currBend_R*Math.sin(curr_radian));       //计算结果的Y分量
1 //计算结果相对于中心点的偏移距离
5 float increase=(float) (currBend_R-currBend_R*Math.cos(curr_radian));
6 //计算结果的XZ分量
7 float result_X=(float) (pointX+increase*Math.sin(Math.toRadians(direction_degree)));
8 float result_Z=(float) (pointZ+increase*Math.cos(Math.toRadians(direction_degree)));
9 position[0]=result_X;                                 //将计算出的位置数据存入结果数组
10    position[1]=result_Y;
11    position[2]=result_Z;
12    position[3]=(float) Math.cos(Math.toRadians(direction_degree));//计算旋转轴的X分量
13    position[4]=(float) Math.sin(Math.toRadians(direction_degree)); /计算旋转轴的Z分量
14    position[5]= (float) Math.toDegrees(curr_radian);        //计算旋转的角度
15    return position;                                          //返回结果数组
16 }
```

- 第3行利用弧长公式计算出当前弯曲半径对应的弧度。
- 第4-8行根据计算出的弧度及风向计算出树叶位置偏移的XYZ分量。
- 第12-14行是根据当前的风力、风向计算出树叶的旋转轴XZ分量以及旋转角度。

（6）最后给出的是绘制树叶的drawSelf方法，其来自TreeLeavesControl类，具体代码如下。

代码位置：见随书光盘中源代码/第13章/Sample13_3/com/bn/Sample13_3目录下的TreeLeavesControl.java。

```
public void drawSelf(int tex_leavesId,float bend_R,float wind_direction){//绘制树叶
  MatrixState.pushMatrix();
  MatrixState.translate(positionX, positionY, positionZ);  //移动到指定的位置
  float curr_height=Constant.leaves_absolute_height;       //当前叶子矩形的绝对高度
  float result[]=resultPoint(wind_direction,bend_R,0,curr_height,0);//计算偏移量和旋转角
  MatrixState.translate(result[0], result[1], result[2]);  //进行偏移
  MatrixState.rotate(result[5], result[3],0,-result[4]);   //进行旋转
  treeLeaves.drawSelf(tex_leavesId);                       //绘制
  MatrixState.popMatrix();
 }
```

提示

此方法根据前面resultPoint方法计算出来的位置偏移数据，旋转轴、旋转角度数据，在绘制树叶前首先对坐标系进行对应的平移，然后再对坐标系进行对应的旋转，最后绘制树叶。

13.4　展翅飞翔的雄鹰

前面3节分别介绍了3个不同的软体的例子，虽然采用的数学模型各有不同，但都是通过编程直接实现特定的数学模型以实现软体动画

的。这在一般的情况下足够用了，但如果想呈现非常复杂的软体动画就很困难了。

有些复杂软体的动画虽然也可以采用数学模型编程实现，但对应的数学模型非常复杂，编程成本很高。本节将给出一种非常简便的实现软体动画的策略——关键帧动画，通过其可以方便地实现游戏中雄鹰飞过蓝天、英雄举刀杀敌的动画。

13.4.1 基本原理

关键帧动画的基本思想非常简单，就是给顶点着色器提供动画中每个关键帧对应的各个顶点的位置数据以及融合比例。顶点着色器根据两套位置数据及当前融合的比例融合出一套结果顶点位置数据。只要在绘制每一帧时提供不同的混合比例即可产生想要的动画。

如本节将要给出的展翅飞翔的雄鹰动画中就用到了3个关键帧，包含4个动画阶段。

- 第一阶段是对1、2号关键帧中的顶点数据进行融合，即从1号关键帧到2号关键帧。
- 第二阶段是对2、3号关键帧中的顶点数据进行融合，即从2号关键帧到3号关键帧。
- 第三阶段是对3、2号关键帧中的顶点数据进行融合，即从3号关键帧到2号关键帧。
- 第四阶段是对2、1号关键帧中的顶点数据进行融合，即从2号关键帧到1号关键帧。

上述4个阶段不断重复就可以呈现出雄鹰展翅飞翔的动画，每个关键帧的具体顶点位置情况如图13-12所示。

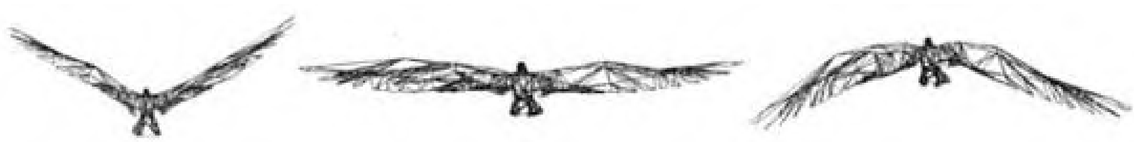


图13-12　雄鹰动画3个关键帧的线框图

说明

从图13-12中可以看出，最左侧的是雄鹰翅膀上扬到最高位置的情况，中间的是雄鹰翅膀放平的情况，右侧是雄鹰翅膀下垂到最低

位置的情况。

到这里读者可能会产生一个疑问：为什么一定要3个关键帧呢？仅保留1、3号关键帧不也能融合出动画吗？确实如此，只保留1、3两个关键帧是可以的，但动画的真实感就会大打折扣。因为仅通过1、3关键帧融合出来的翅膀展平的情况翅膀就会缩短，如图13-13所示。



图13-13　仅保留1、3号关键帧的缺陷

从图13-13中可以看出使用关键帧动画的一个要领，那就是不重要的中间帧可以通过按比例融合两个关键帧得到，真实感基本不受影响。但关键帧不应该省略而通过其他关键帧融合得到，否则动画的真实感就会变差。

提示

使用基于顶点位置的融合的关键帧动画时有一点需要特别注意，那就是所有关键帧中顶点的数量要是一致的，并能够形成一一对应的关系。

13.4.2　开发步骤

上一小节中介绍了关键帧动画的基本原理以及注意事项，本小节将给出一个关键帧动画的案例Sample13_4，其运行效果如图13-14所示。



图13-14　案例Sample13_4的运行效果图

> **说明**
>
> 图13-14中给出了雄鹰展翅飞翔动画中的3帧画面，效果非常真实。由于本书插图是灰度印刷效果可能不是很好，请读者自行用真机运行本案例体会。案例运行时可以用手指在屏幕上滑动以旋转雄鹰从不同的角度观察。

了解了案例的运行效果后，接下来将对本案例的具体开发过程进行简要介绍。由于本案例中的大部分类和前面章节很多案例中的非常类似，因此在这里只给出本案例中比较有代表性的部分，具体内容如下所列。

（1）本案例中用到的雄鹰的3个关键帧是采用3DMax设计并导出成obj文件的，因此首先需要将3个关键帧的顶点数据加载进应用程序并存放到缓冲中，相关代码如下。

代码位置：见随书光盘中源代码/第13章/Sample13_4/com/bn/Sample13_4目录下的GledeForDraw.java。

```
public void initVertexData(MySurfaceView mv) {              //初始化顶点数据的方法
    //加载雄鹰模型
    glede_one=LoadUtil.loadFromFileVertexOnly("laoying01.obj",mv);
    glede_two=LoadUtil.loadFromFileVertexOnly("laoying02.obj",mv)[0];
    glede_three=LoadUtil.loadFromFileVertexOnly("laoying03.obj",mv)[0];
    //创建第一个顶点坐标数据缓冲
```

```
vCount=glede_one.length/3;
ByteBuffer vbb = ByteBuffer.allocateDirect(glede_one[0].length*4);
vbb.order(ByteOrder.nativeOrder());                          //设置字节顺序
    mVertexBuffer1 = vbb.asFloatBuffer();                    //转换为 Float 型缓冲
    mVertexBuffer1.put(glede_one[0]);                        //向缓冲区中放入顶点坐标数据
    mVertexBuffer1.position(0);                              //设置缓冲区起始位置
    //创建第二个顶点坐标数据缓冲
    vCount=glede_two.length/3;
    vbb = ByteBuffer.allocateDirect(glede_two.length*4);
    vbb.order(ByteOrder.nativeOrder());                      //设置字节顺序
    mVertexBuffer2 = vbb.asFloatBuffer();                    //转换为 Float 型缓冲
    mVertexBuffer2.put(glede_two);                           //向缓冲区中放入顶点坐标数据
    mVertexBuffer2.position(0);                              //设置缓冲区起始位置
    //创建第三个顶点坐标数据缓冲
    vCount=glede_three.length/3;
    vbb = ByteBuffer.allocateDirect(glede_three.length*4);
    vbb.order(ByteOrder.nativeOrder());                      //设置字节顺序
    mVertexBuffer3 = vbb.asFloatBuffer();                    //转换为 Float 型缓冲
    mVertexBuffer3.put(glede_three);                         //向缓冲区中放入顶点坐标数据
    mVertexBuffer3.position(0);                              //设置缓冲区起始位置
    //创建纹理坐标数据缓冲
    ByteBuffer tbb = ByteBuffer.allocateDirect(glede_one[1].length*4);
    tbb.order(ByteOrder.nativeOrder());                      //设置字节顺序
    mTexCoorBuffer = tbb.asFloatBuffer();                    //转换为 Float 型缓冲
    mTexCoorBuffer.put(glede_one[1]);                        //向缓冲区中放入顶点纹理坐标数据
    mTexCoorBuffer.position(0);                              //设置缓冲区起始位置
}
```

说明

上述代码主要是将obj文件中的顶点位置与纹理坐标数据加载进来。由于3个关键帧中各个对应顶点的纹理坐标是相同的，因此纹理坐标仅保留了一套。但各个关键帧中对应顶点的位置是不同的，因此顶点数据有3套。

（2）为了在顶点着色器中能够根据比例融合关键帧中的顶点数据，因此需要将融合的比例传入渲染管线。由于有3个关键帧，因此融合比例的取值在0～2之间连续变化。由于将融合比例送入渲染管线的代码非常简单，这里就不再赘述，需要的读者请自行参考光盘中的源代码。

（3）接着需要介绍的是执行顶点融合以产生关键帧动画的顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第13章/Sample13_4/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                    //总变换矩阵
attribute vec3 aPosition;                   //顶点位置(来自1号关键帧)
attribute vec3 bPosition;                   //顶点位置(来自2号关键帧)
attribute vec3 cPosition;                   //顶点位置(来自3号关键帧)
attribute vec2 aTexCoor;                    //顶点纹理坐标
uniform float uBfb;                         //融合比例
varying vec2 vTextureCoord;                 //用于传递给片元着色器的纹理坐标
void main(){
```

```
vec3 tv;                                          //融合后的结果顶点
      if(uBfb<=1.0){     //若融合比例小于等于1，则需要执行的是1、2号关键帧的融合
         tv=mix(aPosition,bPosition,uBfb)
      }else{        //若融合比例大于1，则需要执行的是2、3号关键帧的融合
         tv=mix(bPosition,cPosition,uBfb-1.0);
      }
      gl_Position = uMVPMatrix * vec4(tv,1);    //根据总变换矩阵计算此次绘制此顶点的位置
      vTextureCoord = aTexCoor;                 //将接收的纹理坐标传递给片元着色器
   }
```

说明

上述顶点着色器是实现关键帧动画的核心，其根据传入的融合比例选择对应的两个关键帧进行融合。要注意的是，融合时是调用mix函数完成的，这是为了提高执行效率。实际开发中有些功能既可以采用函数完成也可以自己编程完成，笔者强烈建议直接调用函数完成。这是因为系统的函数在大部分情况下比自己开发的同功能代码片段性能优异。

13.5 二维扭曲

本章前面几节的案例都是在3D空间中对顶点位置进行的变换，但本节将给出一个在2D空间中基于顶点位置变换进行二维扭曲的案例。

提示

本质上可以采用本节介绍的技术扭曲任何2D形状，为了便于介绍，本节将基于等边三角形进行介绍。

13.5.1 基本原理

介绍本节案例的具体开发之前，首先需要了解一下二维扭曲的基本情况，如图13-15所示。

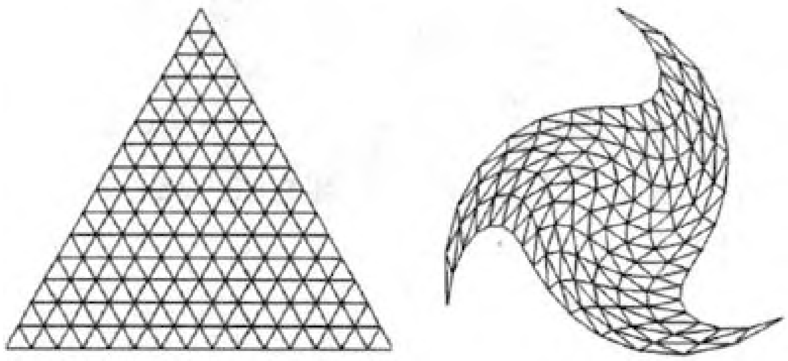


图13-15　扭动的三角形线框图

从图13-15中可以看出，左侧的原始三角形经过扭曲处理后产生了右侧奇异的形状，犹如一个风车。同时从图中可以看出，要想能对原始三角形实现扭曲处理，必须将大三角形切分为很多的小三角形。下面就简单介绍一下扭曲的计算思路（如图13-16所示），具体步骤如下。

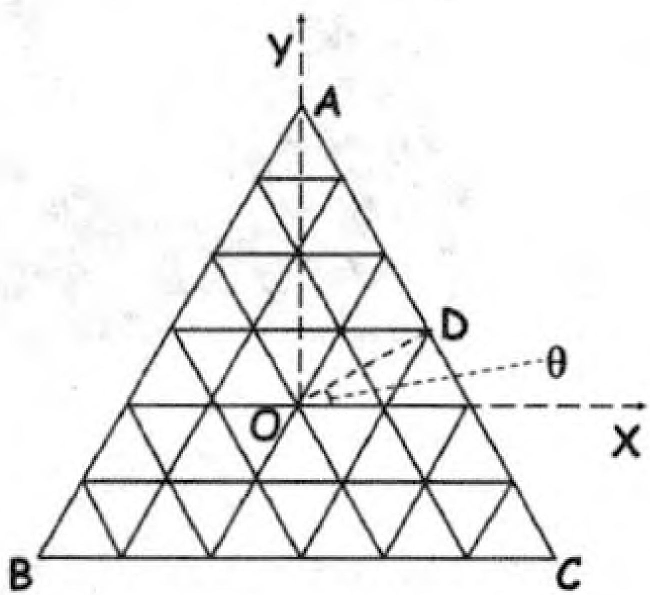


图13-16　扭动的三角形原理图

（1）设扭动的中心点为点O，其坐标为（X0, Y0）；绕中心点被扭动的点为D，其坐标为（X1，Y1）。

（2）设D点在X方向上的偏移为XSpan, Y方向上的偏移为YSpan，则有如下结论。

$$XSpan = X1 - X0$$

$$YSpan = Y1 - Y0$$

$$OD = \sqrt{XSpan \times XSpan + YSpan \times YSpan}$$

（3）接着就可以求出OD与*X* 轴正方向的夹角θ，具体情况如下。

- 如果XSpan=0，并且YSpan大于0，那么θ=π/2。
- 如果XSpan=0，并且YSpan小于0，那么θ=3π/2。
- 如果XSpan不等于0，那么θ=atan（YSpan/XSpan）。

（4）然后计算旋转后的D点与*X* 轴正方向的夹角θ′。

θ′=θ+ratio×OD

说明

其中ratio表示与当前总体旋转角度线性相关的一个系数，用于将距离转化为当前考察点的旋转角度。

（5）计算出旋转后的夹角后，就可以求出旋转后点的*X*、*Y* 坐标了。

X1′=X0+OD×cos（θ′）

Y1′=Y0+OD×sin（θ′）

只需要用顶点着色器实现上述算法即可得到非常漂亮的二维扭曲效果。

13.5.2 开发步骤

上一小节中介绍了二维扭曲的基本原理以及注意事项，本小节将给出一个三角形二维扭曲的案例Sample13_5，其运行效果如图13-17所示。

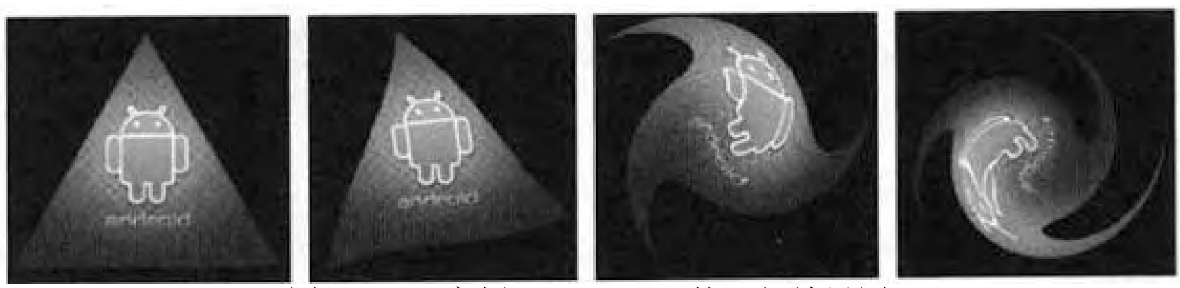


图13-17　案例Sample13_5的运行效果图

了解了案例的运行效果后，接下来将对本案例的具体开发过程进行简要介绍。由于本案例中的大部分类和前面章节很多案例中的非常类似，因此在这里只给出本案例中比较有代表性的部分，具体内容如下所列。

（1）从前面的介绍中已经知道，本案例中的大三角形实际上是由很多小三角形构成的。因此下面首先给出的是自动生成各个小三角形顶点数据的initVertexData方法，其来自MultiTrangle类，具体代码如下。

代码位置：见随书光盘中源代码/第13章/Sample13_5/com/bn/Sample13_5目录下的MultiTrangle.java。

```
public void initVertexData(float edgeLength,int levelNum) {
```

```java
ArrayList<Float> al_vertex=new ArrayList<Float>();           //顶点坐标数据列表
ArrayList<Float> al_texture=new ArrayList<Float>();          //纹理坐标数据列表
float perLength = edgeLength/levelNum;                       //小三角形的边长
for(int i=0;i<levelNum;i++){                                 //循环每一层生成小三角形
    int currTopEdgeNum=i;                                    //当前层顶端边数
    int currBottomEdgeNum=i+1;                               //当前层底端边数
    float currTrangleHeight=(float) (perLength*Math.sin(Math.PI/3)); //每个三角形的高度
    float topEdgeFirstPointX=-perLength*currTopEdgeNum/2;//当前层顶端最左边点的 X 坐标
        float topEdgeFirstPointY=-i*currTrangleHeight;//当前层顶端最左边点的 Y 坐标
        float topEdgeFirstPointZ=0;                    //当前层顶端最左边点的 Z 坐标
        float bottomEdgeFirstPointX=-perLength*currBottomEdgeNum/2;//当前层底端最
        float bottomEdgeFirstPointY=-(i+1)*currTrangleHeight;   //左边点的 X、
        float bottomEdgeFirstPointZ=0;                       //Y、Z 坐标
        float horSpan=1/(float)levelNum;                     //横向纹理的偏移量
        float verSpan=1/(float)levelNum;                     //纵向纹理的偏移量
        float topFirstS=0.5f-currTopEdgeNum*horSpan/2;       //当前层顶端最左边点的
        float topFirstT=i*verSpan;                           //纹理 ST 坐标
        float bottomFirstS=0.5f-currBottomEdgeNum*horSpan/2; //当前层底端最左边点的
        float bottomFirstT=(i+1)*verSpan;                    //纹理 ST 坐标
        for(int j=0;j<currBottomEdgeNum;j++){//循环产生当前层各个上三角形的顶点数据
            float topX=topEdgeFirstPointX+j*perLength;       //当前三角形顶端点的
            float topY=topEdgeFirstPointY;                   //X、Y、Z 坐标
            float topZ=topEdgeFirstPointZ;
            float topS=topFirstS+j*horSpan;                  //当前三角形顶端点的
            float topT=topFirstT;                            //S、T 纹理坐标
            float leftBottomX=bottomEdgeFirstPointX+j*perLength;//当前三角形左下侧
            float leftBottomY=bottomEdgeFirstPointY;         //点的 X、Y、Z 坐标
            float leftBottomZ=bottomEdgeFirstPointZ;
            float leftBottomS=bottomFirstS+j*horSpan;        //当前三角形左下侧
            float leftBottomT=bottomFirstT;                  //点的 S、T 纹理坐标
            float rightBottomX=leftBottomX+perLength;        //当前三角形右下侧
            float rightBottomY=bottomEdgeFirstPointY;        //点的 X、Y、Z 坐标
            float rightBottomZ=bottomEdgeFirstPointZ;
            float rightBottomS=leftBottomS+horSpan;          //当前三角形右下侧
            float rightBottomT=leftBottomT;                  //点的 S、T 纹理坐标
            //将当前三角形顶点数据按照逆时针顺序送入顶点坐标、纹理坐标列表
            al_vertex.add(topX);al_vertex.add(topY);al_vertex.add(topZ);
            al_vertex.add(leftBottomX);al_vertex.add(leftBottomY);al_vertex.add(leftBottomZ);
            al_vertex.add(rightBottomX);al_vertex.add(rightBottomY);al_vertex.add(rightBottomZ);
            al_texture.add(topS);al_texture.add(topT);
            al_texture.add(leftBottomS);al_texture.add(leftBottomT);
            al_texture.add(rightBottomS);al_texture.add(rightBottomT);
        }
        for(int k=0;k<currTopEdgeNum;k++){ //循环产生当前层各个下三角形的顶点数据
            float leftTopX=topEdgeFirstPointX+k*perLength;  //当前三角形左上侧
            float leftTopY=topEdgeFirstPointY;               //点的 X、Y、Z 坐标
            float leftTopZ=topEdgeFirstPointZ;
```

```
            float leftTopS=topFirstS+k*horSpan;              //当前三角形左上侧
            float leftTopT=topFirstT;                        //点的S、T纹理坐标
            float bottomX=bottomEdgeFirstPointX+(k+1)*perLength;//当前三角形底端点
            float bottomY=bottomEdgeFirstPointY;             //的X、Y、Z坐标
            float bottomZ=bottomEdgeFirstPointZ;
            float bottomS=bottomFirstS+(k+1)*horSpan;        //当前三角形右底端点
            float bottomT=bottomFirstT;                      //的S、T纹理坐标
            float rightTopX=leftTopX+perLength;              //当前三角形右上侧
            float rightTopY=leftTopY;                        //点的X、Y、Z坐标
            float rightTopZ=leftTopZ;
            float rightTopS=leftTopS+horSpan;                //当前三角形右上侧
            float rightTopT=topFirstT;                       //点的S、T纹理坐标
            al_vertex.add(leftTopX);al_vertex.add(leftTopY);al_vertex.
            add(leftTopZ); //逆时针卷绕
            al_vertex.add(bottomX);al_vertex.add(bottomY);al_vertex.add(bottomZ);
            al_vertex.add(rightTopX);al_vertex.add(rightTopY);al_vertex.
            add(rightTopZ);
            al_texture.add(leftTopS);al_texture.add(leftTopT);
            al_texture.add(bottomS);al_texture.add(bottomT);
            al_texture.add(rightTopS);al_texture.add(rightTopT);
        }}
        ……//此处省略部分代码，需要的读者请参考光盘
}
```

说明

上述代码根据传入的大三角形边长及分层数量自动计算出了每一层中各个三角形的顶点坐标、纹理坐标。每一层中的三角形分为上三角形与下三角形，顶点计算方法不同，如图13-18所示。

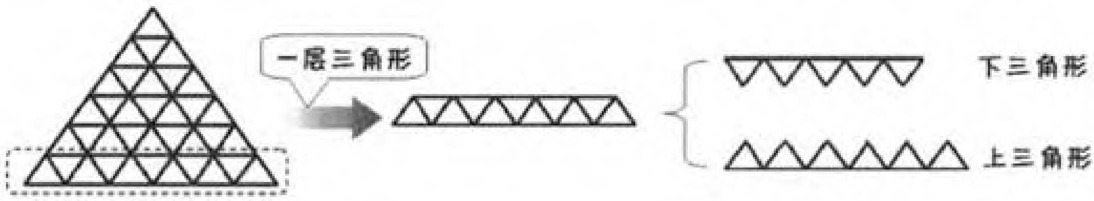


图13-18　每层的上三角形与下三角形

（2）要能够呈现出扭动的三角形就需要在绘制每一帧时将不同的、连续变化的整体扭动角度因子（就是上一小节算法中的ratio）传入渲染管线。由于这部分代码非常简单，这里不再赘述，需要的读者请参考随书光盘。

（3）接着介绍的是实现二维扭曲算法的顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第13章/Sample13_5/assets目录下的vertex_tex.sh。

```
uniform mat4 uMVPMatrix;                    //总变换矩阵
attribute vec3 aPosition;                   //顶点位置
attribute vec2 aTexCoor;                    //顶点纹理坐标
varying vec2 vTextureCoord;                 //用于传递给片元着色器的纹理坐标
uniform float ratio;                        //当前整体扭动角度因子
void main(){
```

```
float pi = 3.1415926;                                  //圆周率
float centerX=0.0;                                     //中心点X坐标
    float centerY=-5.0;                                //中心点Y坐标
    float currX = aPosition.x;                         //当前点X坐标
    float currY = aPosition.y;                         //当前点Y坐标
    float spanX = currX - centerX;                     //当前X偏移量
    float spanY = currY - centerY;                     //当前Y偏移量
    float currRadius = sqrt(spanX * spanX + spanY * spanY);  //计算距离
    float currRadians;                        //当前点与X轴正方向的夹角，用弧度表示
    if(spanX != 0.0) {                        //计算当前点与X轴正方向的夹角
        currRadians = atan(spanY , spanX); //一般情况
    }else{
        currRadians = spanY > 0.0 ? pi/2.0 : 3.0*pi/2.0; //0.5π和1.5π的特殊情况
    }
    float resultRadians = currRadians + ratio*currRadius;    //计算出扭曲后的角度
    float resultX = centerX + currRadius * cos(resultRadians); //计算结果点的X坐标
    float resultY = centerY + currRadius * sin(resultRadians);//计算结果点的Y坐标
    //构造结果点，并根据总变换矩阵计算此次绘制此顶点的位置
   gl_Position = uMVPMatrix * vec4(resultX,resultY,0.0,1);
   vTextureCoord = aTexCoor;               //将接收的纹理坐标传递给片元着色器
}
```

说明

上述顶点着色器实现了上一小节介绍的二维扭曲算法，通过应用此顶点着色器可以对任意的二维物体进行扭曲，并不一定是三角形。

进行二维扭曲效果的开发时，有一个需要特别注意的地方。那就是提供给渲染管线的模型一定要分得比较细，如果分得比较粗糙，最终的结果就会比较差，如图13-19所示。

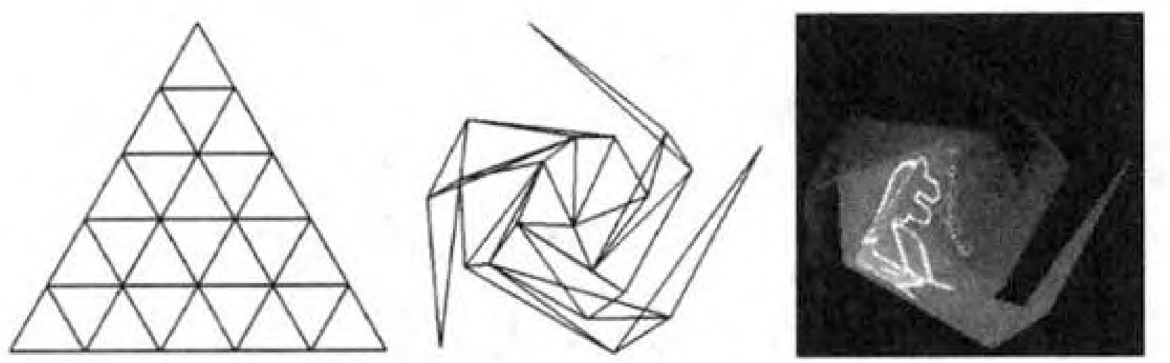


图13-19　模型切分得很粗导致扭曲效果很差

> **说明**
>
> 图13-19给出了将本案例中的大三角形切分得比较粗糙后的运行效果以及线框图。从图中可以看出，切分得很粗糙后实际扭曲的效果与期望的效果之间就有较大的差距了。

13.6　OpenGL ES 1.x与OpenGL ES 2.0实现方案的对比

本章最开始提到过，在OpenGL ES 1.x平台想高效实现本章的案例是非常困难的。这是因为OpenGL ES 1.x采用的是固定渲染管线，顶点数据一旦送入管线后就不可能对其进行自定义的处理了。因此，在OpenGL ES 1.x时代想实现本章软体或扭曲的效果只能采用如下两种策略之一。

1．初始化时预先计算数据

此种策略的基本思想非常简单，就是在初始化时将动画中每一帧画面的顶点数据都计算出来，绘制每一帧画面时将与此帧画面对应的

顶点数据送入渲染管线即可。因此，此策略就有两方面明显的局限性。

- 动画中的帧数不能过多，否则会占用大量的内存，导致程序不能正常执行。由于帧数不是很多，因此对于动态范围大的动画就会不太平滑，效果一般。
- 动画中的每一帧是预先计算好的，不能够根据用户交互或情况的变化而变化，灵活性差。

2．绘制每帧画面前由CPU临时计算顶点数据

此种策略其实就是将本章案例中由顶点着色器完成的工作改为绘制每帧画面前由CPU来完成，其可以解决第一种策略的两个局限，但本身也有很大的局限性。因为在此情况下，CPU一方面需要处理顶点数据，同时还承载了很多处理其他业务逻辑的任务，如人机交互、物理碰撞等，这会导致CPU不堪重负，整个程序运行很慢。

从上述OpenGL ES 1.x时代实现软体或扭曲动画的策略中可以看出，基于OpenGL ES 1.x很难完全发挥出GPU强大的处理能力，因此整个3D开发产业现在都在向OpenGL ES 2.0迈进。读者在以后的开发中也可以多思考、多总结，将能够由着色器完成的工作都让着色器去完成，尽量把CPU解放出来。

> **提示**
>
> 笔者也是从OpenGL ES 1.x时代走过来的，那时开发出的一些高级的特效，一方面代码很长，开发成本高。另一方面，运行速度慢，不得不做出很多牺牲。现在有了OpenGL ES 2.0，原来的很多限制都不复存在了，相信随着硬件的进一步发展，可以开发出更多、更好的3D特效。

13.7　本章小结

本章主要介绍了顶点着色器的几种巧妙用法，并通过典型的案例实现了这些用法。通过本章的学习，读者可以进一步领会到顶点着色器的强大作用，为以后开发更加酷炫的3D应用打下坚实的基础。

第14章

片元着色器的妙用

上一章介绍了顶点着色器的妙用，给出了不少顶点着色器的使用技巧。其实片元着色器在开发中一样可以大显身手，本章将通过几个非常实用的案例来介绍这方面的知识。

14.1　程序纹理技术

前面章节的案例中对物体进行纹理映射时采用的基本都是预先由美工人员设计好的纹理图，这种模式能满足大部分的需要。但需要满足一些特殊的效果时，也可以使用片元着色器程序来基于一定的规则计算每个片元的颜色，这就是程序纹理技术。

提示

其实前面第6章介绍光照时对圆球采用的棋盘着色器就是基于程序纹理技术的，本节将再给出两个这方面的案例。

14.1.1 砖块着色器

本小节将介绍本节的第一个程序纹理着色器——砖块着色器，其基本原理如图14-1所示。

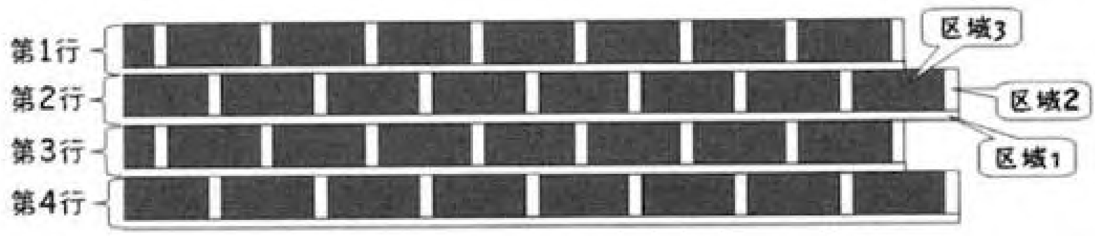


图14-1 砖块着色器原理图

从图14-1中可以看出，砖块着色器可以实现类似于砖墙的效果，具体的实现细节如下。

（1）先根据需着色片元的某种参数计算出片元位于哪一行（纵向分割），并记录下行号。

（2）再根据片元的参数计算出片元是否位于此行的区域1中（纵向分割），若位于区域1中则采用砖块缝隙的水泥色着色。

（3）若片元不在区域1中，则根据行号的奇偶性及片元的参数计算出片元是否位于此行的区域3中（横向分割）。若位于区域3中则采用砖块色着色，否则也采用砖块缝隙的水泥色着色。之所以需要依据行号的奇偶性是因为奇数行与偶数行要偏移半个砖块。

提示

进行纵向分割时的依据有很多选择，如本小节案例是对球面进行的，则可以采用纬度。若表面是平面，则可以采用某个轴的坐标。进行横向分割时的依据也有很多选择，如本小节案例是对球面

进行的，则可以采用经度。若表面是平面，则也可以采用某个轴的坐标。

了解了砖块着色器的基本原理后，下面请读者了解一下本小节案例Sample14_1的运行效果，如图14-2所示。

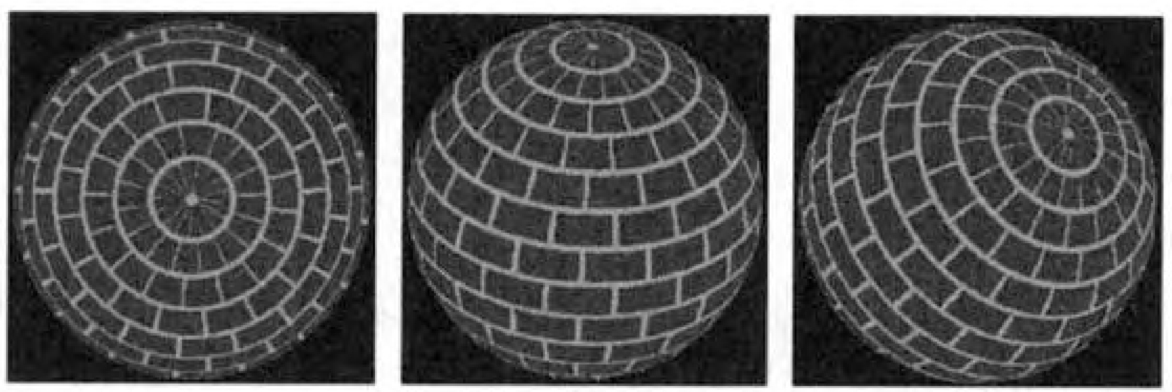


图14-2　案例Sample14_1的运行效果图

从图14-2中可以看出，本小节案例是针对球面应用的砖块着色器。由于构建球面的知识在前面的章节中已经详细介绍过，故这里仅给出与砖块着色器相关的部分代码，具体内容如下。

（1）首先在负责绘制球面的Ball类中，需要增加将顶点经纬度数据传入渲染管线的相关代码，此部分代码与将顶点坐标传递进渲染管线的代码套路完全一致。因此这里不再赘述，需要的读者请参考随书光盘中的源代码。

（2）接下来给出的是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第14章/Sample14_1/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                    //总变换矩阵
attribute vec3 aPosition;                   //顶点位置
attribute vec2 aLongLat;                    //顶点经纬度
varying vec2 mcLongLat;                     //用于传递给片元着色器的顶点经纬度
void main() {
  gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点的位置
  mcLongLat=aLongLat;                 //将顶点的经纬度传递给片元着色器
}
```

说明

与普通的顶点着色器相比，上述顶点着色器中主要是增加了接收从管线传入的顶点经纬度，并将其传递给片元着色器的相关代码。

（3）最后给出的是片元着色器，其代码如下。

代码位置：见随书光盘中源代码/第14章/Sample14_1/assets目录下的frag.sh。

```
precision mediump float;                    //给出默认的浮点精度
varying vec2 mcLongLat;                     //从顶点着色器传递过来的经纬度
void main() {
  vec3 bColor=vec3(0.678,0.231,0.129);      //砖块的颜色
```

```
vec3 mColor=vec3(0.763,0.657,0.614);              //水泥的颜色
vec3 color;                                       //片元的最终颜色
int row=int(mod((mcLongLat.y+90.0)/12.0,2.0));    //计算当前片元位于奇数行还是偶数行
float ny=mod(mcLongLat.y+90.0,12.0);        //计算当前片元是否在此行区域1中的辅助变量
float oeoffset=0.0;                         //每行的砖块偏移值，奇数行偏移半个砖块
    float nx;                               //当前片元是否在此行区域3中的辅助变量
    if(ny>10.0) {                           //位于此行的区域1中
      color=mColor;                         //采用水泥色着色
    } else {                                //不位于此行的区域1中
      if(row==1) {                          //若为奇数行则偏移半个砖块
        oeoffset=11.0;
      }
      nx=mod(mcLongLat.x+oeoffset,22.0);    //计算当前片元是否在此行区域3中的辅助变量
      if(nx>20.0) {                         //不位于此行的区域3中
        color=mColor;                       //采用水泥色着色
      } else {                              //位于此行的区域3中
        color=bColor;                       //采用砖块色着色
    }}
    gl_FragColor=vec4(color,0);             //将片元的最终颜色传递进管线
 }
```

说明

上述片元着色器按照前面介绍的砖块着色器的原理实现了针对球面的砖块着色器，分割的依据是片元的经纬度。有兴趣的读者还可以针对其他表面基于不同的分割依据实现砖块着色器，以进一步加深理解。

14.1.2 沙滩球着色器

上一小节介绍了砖块着色器，本小节将介绍另一种基于程序纹理技术的着色器——沙滩球着色器。沙滩球着色器的实现相比于砖块着色器要简单不少，主要思路为，将靠近球面两级的片元用白色着色，其他部分按照经度切分，以不同的颜色着色。

介绍本小节案例Sample14_2所使用的着色器前，首先请读者了解一下本小节案例的运行效果，如图14-3所示。

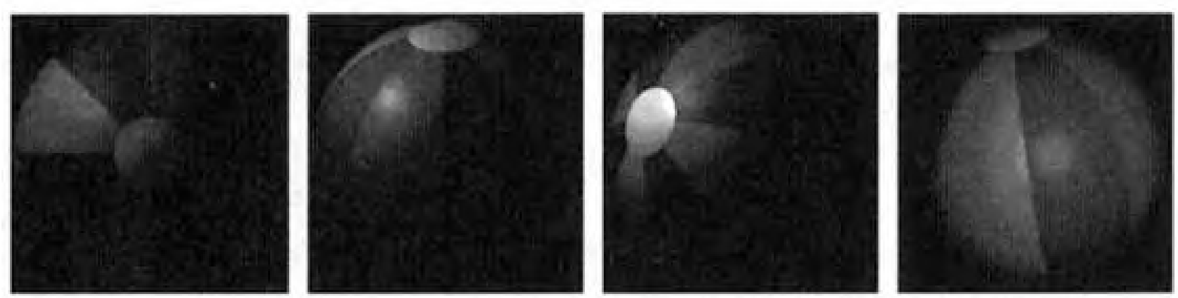


图14-3 案例Sample14_2的运行效果图

了解了案例的运行效果后，下面就可以进行案例的开发了。由于本案例中的大部分代码与上一小节的案例非常相似，主要的区别在于片元着色器。因此这里仅给出本案例中的片元着色器，其代码如下。

代码位置：见随书光盘中源代码/第14章/Sample14_2/assets目录下的frag.sh。

```
precision mediump float;                          //给出默认的浮点精度
varying vec4 vAmbient;                            //从顶点着色器传递过来的环境光分量
varying vec4 vDiffuse;                            //从顶点着色器传递过来的散射光分量
varying vec4 vSpecular;                           //从顶点着色器传递过来的镜面光分量
varying vec2 mcLongLat;                           //从顶点着色器传递过来的经纬度
void main() {
  vec3 color;                                     //片元的最终颜色
  if(abs(mcLongLat.y)>75.0){
      color = vec3(1.0,1.0,1.0);                  //两极附近是白色
   }else{
      int colorNum = int(mcLongLat.x/45.0);       //根据经度计算出颜色编号
      if(colorNum == 0){
          color = vec3(1.0,0.0,0.0);              //0 号颜色
      }else if(colorNum == 1){
          color = vec3(0.0,1.0,0.0);              //1 号颜色
      }else if(colorNum == 2){
          color = vec3(0.0,0.0,1.0);              //2 号颜色
      }else if(colorNum == 3){
          color = vec3(1.0,1.0,0.0);              //3 号颜色
      }else if(colorNum == 4){
          color = vec3(1.0,0.0,1.0);              //4 号颜色
      }else if(colorNum == 5){
          color = vec3(0.0,1.0,1.0);              //5 号颜色
      }else if(colorNum == 6){
          color = vec3(0.3,0.4,0.7);              //6 号颜色
      }else if(colorNum == 7){
          color = vec3(0.3,0.7,0.2);              //7 号颜色
   }}
   vec4 finalColor = vec4(color,1.0);         //将颜色扩充为带 Alpha 通道的 Vec4 类型
   //综合 3 个通道光的最终强度及片元的颜色计算出最终片元的颜色并传递给管线
   gl_FragColor=finalColor*vAmbient + finalColor*vDiffuse + finalColor*vSpecular;
}
```

- 第8-9行中首先判断片元是否靠近两极，如果靠近两极，就采用白色对片元着色。
- 第11-27行首先将不靠近两极的片元根据经度确定其位于的颜色区域，然后采用对应区域的既定颜色对片元着色。

> **提示**
>
> 相比于上一小节中砖块着色器的案例，本案例中还引入了光照。光照部分的相关代码与第6章介绍光照时案例中的完全一致，需要的读者请参考随书光盘中的源代码。

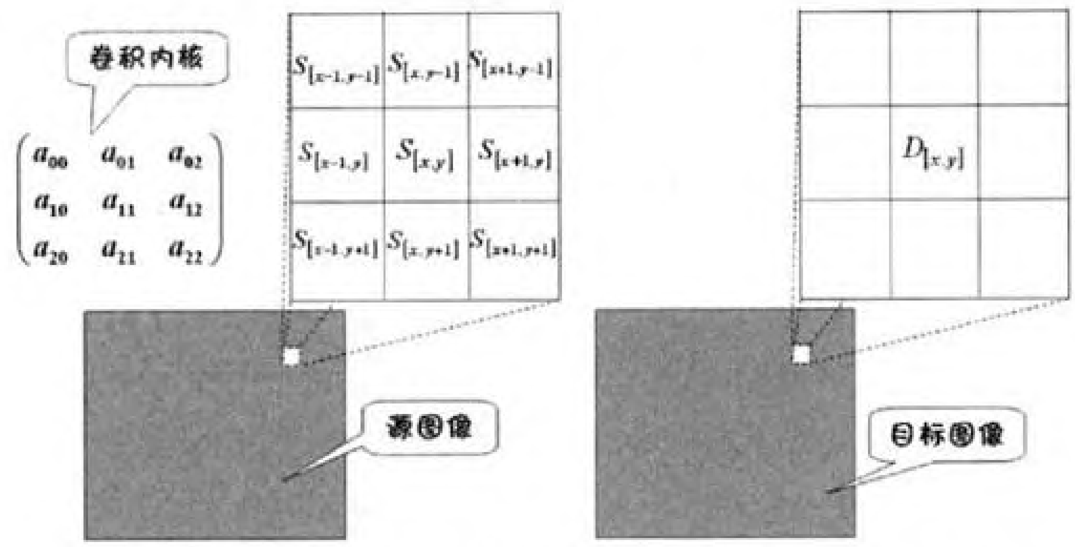
14.2 数字图像处理

随着数码相片逐渐替代传统胶卷相片成为主流，数字图像处理与我们生活的关系也越来越密切了。例如经常用来进行相片后期制作的PhotoShop中就带有很多特效滤镜，如：模糊、浮雕、锐化等，这些都是基于数字图像处理技术开发的。

传统的数字图像处理应用大部分是基于CPU开发的，由于CPU运算能力的限制，处理一幅图片需要较长的时间。本节将介绍的数字图像处理应用是基于GPU开发的，具有开发简便，速度快等优势。

14.2.1 卷积的基本知识

数字图像处理中用到的很多滤镜都是基于卷积计算开发的，因此本小节将对卷积计算的原理进行简要的介绍。卷积是一种很常见的数字图像处理操作，其可以用来过滤一幅图像。实现过滤的方法是计算源图像与卷积内核之间的积，所谓卷积内核是指一个nxn的矩阵，n一般为奇数。进行卷积计算时将卷积内核对待处理图像中的每个像素都应用一次，具体的计算思路如图14-4所示。



$$D_{[x,y]} = S_{[x-1,y-1]} \times a_{00} + S_{[x,y-1]} \times a_{01} + S_{[x+1,y-1]} \times a_{02} + \cdots + S_{[x+1,y+1]} \times a_{22}$$

图14-4　卷积基本原理

从图14-4中可以看出，卷积计算将需要处理的图片中的每个像素计算一次，具体的计算方法如下。

- 首先将卷积内核中心的元素对准当前待处理的像素，此时卷积内核中的其他元素也各自对应到了一个像素。
- 然后将卷积内核中的各元素与其对应的像素的颜色值相乘。
- 最后将所有的乘积加权求和即可得到处理后图片中此位置像素的颜色值。

根据卷积内核中个各元素值的不同，可以轻松地实现模糊、边缘检测、锐化、浮雕等滤镜的特效，下面的小节将陆续给出这方面的案例。

提示

请读者注意的是，卷积内核的尺寸并不一定是3×3，也可以是5×5、7×7等。但卷积内核越大，计算量成几何级数增长，因此采用3×3的情况较多。

14.2.2　平滑过滤

平滑过滤可以将过于锐利的照片变得平滑些，实现起来也非常简单，只需要将卷积内核的所有元素值都设为1即可，下面的表14-1给出了本节案例所采用的3×3平滑过滤的卷积内核。

表14-1　实现平滑过滤的卷积内核

1	1	1
1	1	1
1	1	1

了解了平滑过滤的卷积内核后，下面请读者进一步了解一下本小节案例Sample14_3的运行效果，如图14-5所示。

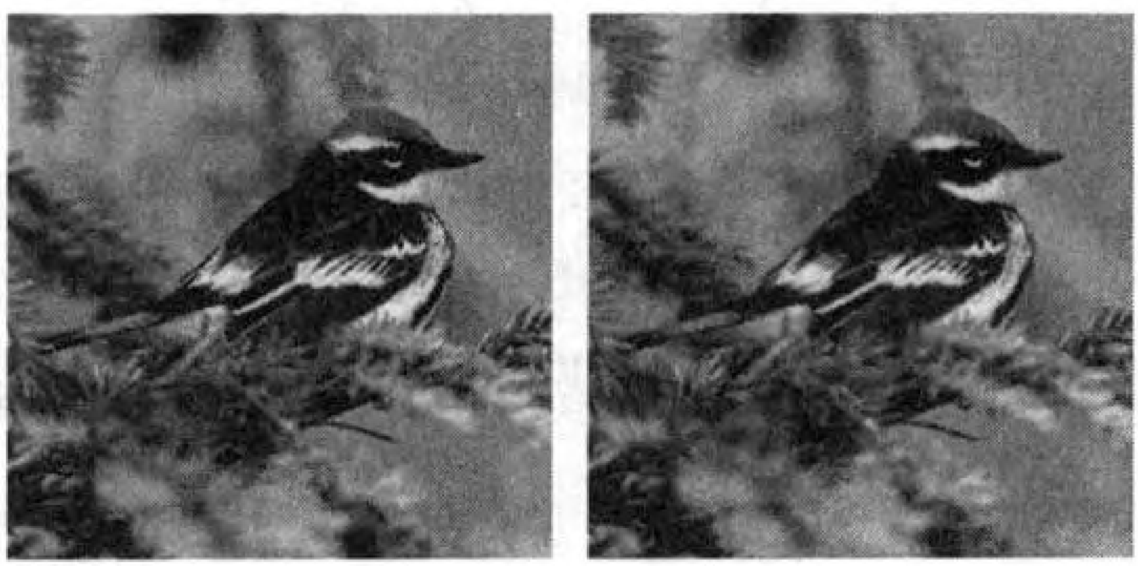


图14-5　案例Sample14_3的运行效果图

说明

图14-5中左侧为处理前的原图，右侧为经过平滑过滤的图像。细致比对左右两幅图可以看出，右侧的图片比左侧的要平滑了不少。由于插图灰度印刷的原因，读者也可能看得不是很清楚，此时请读者自行在真机上运行本案例观察。

了解了案例所采用的卷积内核与案例的运行效果后，就可以进行案例的开发了。由于本案例中的大部分代码在前面章节的许多案例中都出现过，故这里仅给出实现卷积的片元着色器，其代码如下。

代码位置：见随书光盘中源代码/第14章/Sample14_3/assets目录下的frag.sh。

```
precision mediump float;                //给出默认的浮点精度
varying vec2 vTextureCoord;             //从顶点着色器传递过来的纹理坐标
uniform sampler2D sTexture;             //纹理内容数据
void main() {
    //给出卷积内核中各个元素对应像素相对于待处理像素的纹理坐标偏移量
    vec2 offset0=vec2(-1.0,-1.0); vec2 offset1=vec2(0.0,-1.0); vec2 offset2=vec2(1.0,-1.0);
    vec2 offset3=vec2(-1.0,0.0); vec2 offset4=vec2(0.0,0.0); vec2 offset5=vec2(1.0,0.0);
    vec2 offset6=vec2(-1.0,1.0); vec2 offset7=vec2(0.0,1.0); vec2 offset8=vec2(1.0,1.0);
    const float scaleFactor = 1.0/9.0;//给出最终求和时的加权因子(为调整亮度)
        //卷积内核中各个位置的值
        float kernelValue0 = 1.0; float kernelValue1 = 1.0; float kernelValue2 = 1.0;
        float kernelValue3 = 1.0; float kernelValue4 = 1.0; float kernelValue5 = 1.0;
        float kernelValue6 = 1.0; float kernelValue7 = 1.0; float kernelValue8 = 1.0;
        vec4 sum;//最终的颜色和
        //获取卷积内核中各个元素对应像素的颜色值
        vec4 cTemp0,cTemp1,cTemp2,cTemp3,cTemp4,cTemp5,cTemp6,cTemp7,cTemp8;
        cTemp0=texture2D(sTexture, vTextureCoord.st + offset0.xy/512.0);
        cTemp1=texture2D(sTexture, vTextureCoord.st + offset1.xy/512.0);
        cTemp2=texture2D(sTexture, vTextureCoord.st + offset2.xy/512.0);
        cTemp3=texture2D(sTexture, vTextureCoord.st + offset3.xy/512.0);
        cTemp4=texture2D(sTexture, vTextureCoord.st + offset4.xy/512.0);
        cTemp5=texture2D(sTexture, vTextureCoord.st + offset5.xy/512.0);
        cTemp6=texture2D(sTexture, vTextureCoord.st + offset6.xy/512.0);
        cTemp7=texture2D(sTexture, vTextureCoord.st + offset7.xy/512.0);
        cTemp8=texture2D(sTexture, vTextureCoord.st + offset8.xy/512.0);
```

```
    //颜色求和
    sum =kernelValue0*cTemp0+kernelValue1*cTemp1+kernelValue2*cTemp2+
         kernelValue3*cTemp3+kernelValue4*cTemp4+kernelValue5*cTemp5+
        kernelValue6*cTemp6+kernelValue7*cTemp7+kernelValue8*cTemp8;
    gl_FragColor = sum * scaleFactor; //进行亮度加权后将最终颜色传递给管线
}
```

提示

上述片元着色器的代码使用平滑过滤的卷积内核进行卷积计算，实现了平滑过滤效果的滤镜。同时上述代码中使用的加权因子是用来调节结果图像亮度的，读者可以根据具体情况进行修改，其值越大结果图像亮度越大。

14.2.3　边缘检测

通过卷积不但可以对图像进行平滑处理，还可以进行边缘检测。实现边缘检测的卷积内核各元素的值如表14-2所示。

表14-2　实现边缘检测的卷积内核

0	1	0
1	-4	1
0	1	0

了解了边缘检测的卷积内核后，下面请读者进一步了解一下本小节案例Sample14_4的运行效果，如图14-6所示。



图14-6　案例Sample14_4的运行效果图

说明

图14-6中左侧为处理前的原图，中间为经过边缘检测后的图像。对比两幅图可以看出，中间的图像中仅保留了左侧原图中物体边缘的位置内容。由于插图灰度印刷的原因，读者也可能看的不是很清楚，因此最右边还给出了将结果图像颜色置反后的参考图像，这样看起来会清楚很多。

了解了案例所采用的卷积内核与案例的运行效果后，就可以进行案例的开发了。由于本案例中的大部分代码与上一小节案例中的基本一致，有区别的部分主要就是片元着色器中卷积内核各个元素的值以及控制亮度的加权因子。故这里仅给出有区别部分的代码，具体内容如下。

代码位置：见随书光盘中源代码/第14章/Sample14_4/assets目录下的frag.sh。

```
const float scaleFactor = 0.9;//给出最终求和时的加权因子(为调整亮度)
```

```
//卷积内核中各个位置的值
float kernelValue0 = 0.0; float kernelValue1 = 1.0; float kernelValue2 = 0.0;
float kernelValue3 = 1.0; float kernelValue4 = -4.0; float kernelValue5 = 1.0;
float kernelValue6 = 0.0; float kernelValue7 = 1.0; float kernelValue8 = 0.0;
```

14.2.4 锐化处理

对于过于平滑已经显得模糊的图像也可以使用卷积进行锐化处理，使得图像看起来清晰些。实现锐化处理的卷积内核各元素的值如表14-3所列。

表14-3 实现锐化的卷积内核

0	-1	0
-1	5	-1
0	-1	0

了解了锐化处理的卷积内核后，下面请读者进一步了解一下本小节案例Sample14_5的运行效果，如图14-7所示。

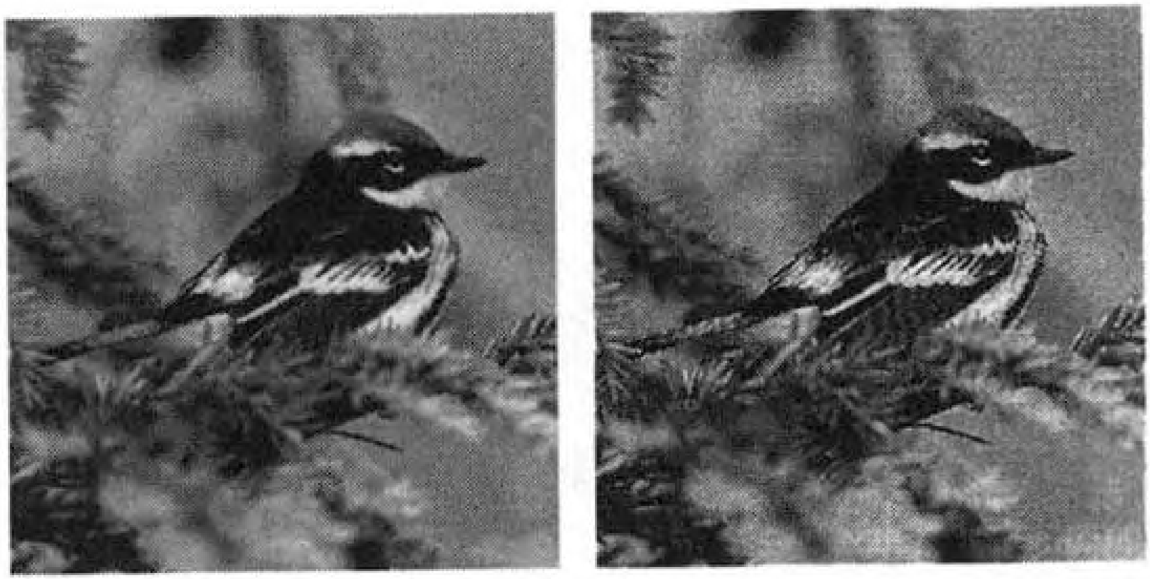


图14-7　案例Sample14_5的运行效果图

说明

图14-7中左侧为处理前的原图，右侧为经过锐化处理后的图像。细致比对左右两幅图可以看出，右侧的图片比左侧的要清晰了一些。由于插图灰度印刷的原因，读者也可能看的不是很清楚，此时请读者自行在真机上运行本案例观察。

看完案例的运行效果后，接下来对本案例的具体开发过程进行介绍。由于本案例只是将案例Sample14_4的片元着色器做了简单的修改，因此这里仅给出修改部分的代码。

代码位置：见随书光盘中源代码/第14章/Sample14_5/assets目录下的frag.sh。

```
const float scaleFactor = 1.0;//给出最终求和时的加权因子(为调整亮度)
//卷积内核中各个位置的值
float kernelValue0 = 0.0; float kernelValue1 = -1.0; float kernelValue2 = 0.0;
float kernelValue3 = -1.0; float kernelValue4 = 5.0; float kernelValue5 = -1.0;
float kernelValue6 = 0.0; float kernelValue7 = -1.0; float kernelValue8 = 0.0;
```

提示

从上述代码中可以看出，主要是修改了片元着色器中卷积内核的元素值以及亮度加权因子，其他部分基本没有变化。

14.2.5 浮雕效果

采用卷积计算还可以产生浮雕效果，实现浮雕效果的卷积内核各元素的值如表14-4所列。

表14-4 实现雕效果的卷积内核

2	0	2
0	0	0
3	0	-6

了解了浮雕效果的卷积内核后，下面请读者进一步了解一下本小节案例Sample14_5A的运行效果，如图14-8所示。

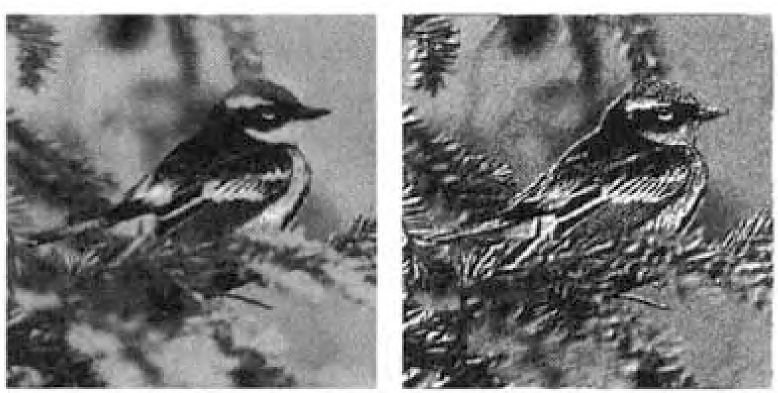


图14-8 案例Sample14_5A的运行效果图

说明

图14-8中左侧为处理前的原图，右侧为经过浮雕处理后的图像。对比左右两幅图可以明显地看出，左侧的原图给人的感觉是平面的，而右侧经过浮雕处理后的结果图像有明显的雕刻的凹凸感。

看完案例的运行效果后，接下来对本案例的具体开发过程进行介绍。由于本案例只是将案例Sample14_5的片元着色器做了简单的修改，因此这里仅给出修改后片元着色器的代码。

代码位置：见随书光盘中源代码/第14章/Sample14_5/assets目录下的frag.sh。

```
precision mediump float;            //给出默认的浮点精度
varying vec2 vTextureCoord;         //从顶点着色器传递过来的纹理坐标
uniform sampler2D sTexture;         //纹理内容数据
void main() {
    //给出卷积内核中各个元素对应像素相对于待处理像素的纹理坐标偏移量
    vec2 offset0=vec2(-1.0,-1.0); vec2 offset1=vec2(0.0,-1.0); vec2 offset2=vec2(1.0,-1.0);
    vec2 offset3=vec2(-1.0,0.0); vec2 offset4=vec2(0.0,0.0); vec2 offset5=vec2(1.0,0.0);
    vec2 offset6=vec2(-1.0,1.0); vec2 offset7=vec2(0.0,1.0); vec2 offset8=vec2(1.0,1.0);
    const float scaleFactor = 1.0;//给出最终求和时的加权因子(为调整亮度)
        //卷积内核中各个位置的值
        float kernelValue0 = 2.0; float kernelValue1 = 0.0; float kernelValue2 = 2.0;
        float kernelValue3 = 0.0; float kernelValue4 = 0.0; float kernelValue5 = 0.0;
        float kernelValue6 = 3.0; float kernelValue7 = 0.0; float kernelValue8 = -6.0;
        vec4 sum;//最终的颜色和
        //获取卷积内核中各个元素对应像素的颜色值
        vec4 cTemp0,cTemp1,cTemp2,cTemp3,cTemp4,cTemp5,cTemp6,cTemp7,cTemp8;
        cTemp0=texture2D(sTexture, vTextureCoord.st + offset0.xy/512.0);
        cTemp1=texture2D(sTexture, vTextureCoord.st + offset1.xy/512.0);
        cTemp2=texture2D(sTexture, vTextureCoord.st + offset2.xy/512.0);
        cTemp3=texture2D(sTexture, vTextureCoord.st + offset3.xy/512.0);
        cTemp4=texture2D(sTexture, vTextureCoord.st + offset4.xy/512.0);
        cTemp5=texture2D(sTexture, vTextureCoord.st + offset5.xy/512.0);
        cTemp6=texture2D(sTexture, vTextureCoord.st + offset6.xy/512.0);
        cTemp7=texture2D(sTexture, vTextureCoord.st + offset7.xy/512.0);
        cTemp8=texture2D(sTexture, vTextureCoord.st + offset8.xy/512.0);
        //颜色求和
        sum =kernelValue0*cTemp0+kernelValue1*cTemp1+kernelValue2*cTemp2+
              kernelValue3*cTemp3+kernelValue4*cTemp4+kernelValue5*cTemp5+
            kernelValue6*cTemp6+kernelValue7*cTemp7+kernelValue8*cTemp8;
        //灰度化
        float hd=(sum.r+sum.g+sum.b)/3.0;
        gl_FragColor = vec4(hd) * scaleFactor; //进行亮度加权后将最终颜色传递给管线
}
```

• 第11-13行中卷积内核各个元素的值与修改前案例Sample14_5中的有很大的不同。

• 第31行对滤镜处理后的颜色进行了灰度化处理，将3个色彩通道值的平均值作为结果颜色各个色彩通道的值。

提示

到这里为止本书需要介绍的基于卷积计算的数字图像处理滤镜就介绍完了。但实际可用的滤镜千变万化，远不止这几个，有兴趣的读者可以查阅数字图像处理相关的技术资料，仿造上述几个案例自行实现更加酷炫的滤镜。

14.2.6 图像渐变

采用片元着色器不但可以轻松开发出各种滤镜，还可以开发出很多有趣的应用。如本小节将给出的图像渐变的例子就是如此，此应用运行时从一幅照片平滑地过渡为另一幅照片，非常有意思。在介绍案例的开发之前，请读者先了解一下案例的运行效果，如图14-9所示。



图14-9　案例Sample14_6的运行效果图

说明

图14-9中从左至右为从一幅照片向另一幅照片平滑过渡的过程。从最左边我们开发团队仇磊的照片平滑过渡到了我们团队夏学良的照片，实现的基本思路就是在不同的时间点采用不同的加权因子对两幅照片进行混合。

了解了案例的运行效果后，下面来介绍本小节案例的开发。由于本小节案例中的大部分代码与前面很多案例中的基本一致，因此这里仅给出本案例中有代表性和特色的部分，具体如下所列。

（1）首先需要在应用程序中定时将连续变化的混合比例因子以及两幅纹理图传入渲染管线，以备片元着色器使用。由于将混合比例因子传入渲染管线的代码与将其他数据传入渲染管线的代码基 本相同，因此这里不再赘述，需要的读者请参考随书光盘中的源代码。

（2）本案例的顶点着色器与普通的纹理映射顶点着色器基本相同，但片元着色器有所不同。因此下面给出本案例中的片元着色器，其代码如下。

代码位置：见随书光盘中源代码/第14章/Sample14_6/assets目录下的frag.sh。

```
precision mediump float;                                     //给出默认的浮点精度
varying vec2 vTextureCoord;                                  //从顶点着色器传递过来的纹理坐标
uniform sampler2D sTexture1;                                 //纹理内容数据1(仇磊照片)
uniform sampler2D sTexture2;                                 //纹理内容数据2(夏学良照片)
uniform float uT;                                            //混合比例因子
void main() {
   vec4 color1 = texture2D(sTexture1, vTextureCoord);        //从纹理1中采样出颜色值1
   vec4 color2 = texture2D(sTexture2, vTextureCoord);        //从纹理2中采样出颜色值2
   gl_FragColor = color1*(1.0-uT) + color2*uT;               //按比例混合两个颜色值
 }
```

说明

上述片元着色器其实非常简单，根据传入的混合比例因子将从两幅纹理图中采样得到的颜色按比例进行混合。只要混合比例因子定时变化，就自然会产生平滑过渡的效果了。

14.3　分形着色器

最近一些年有一门非常热门的几何学分支——分形几何，基于分形几何可以渲染出很多绚丽多彩的图案。本节将基于曼德布罗集

（Mandelbrot set）和茱莉亚集（Julia Set）向读者简单介绍如何基于分形几何开发出具有吸引力的程序纹理着色器。

14.3.1　曼德布罗集简介

曼德布罗集是基于复数在复平面上迭代产生的，因此下面将首先简要介绍一下复数及复平面的知识。每个复数由两部分组成：实部和虚部。实部的基本单位是实数1，虚部的基本单位是i。i是一个很特殊的虚数，其是-1的平方根，即$i^2=-1$。

复数可以用a+bi的基本格式来表示，下面给出了两个复数相乘的规则。

x=a+bi　y=c+di　xy=ac+adi+cbi-bd=(ac-bc)+(ad+bc)i

因为复数包含两个部分，所以每个复数都可以看做是二维平面上的一个点，用实部作为一个轴的坐标，虚部作为另一个轴的坐标。这个平面就称之为复平面，具体情况如图14-10所示。

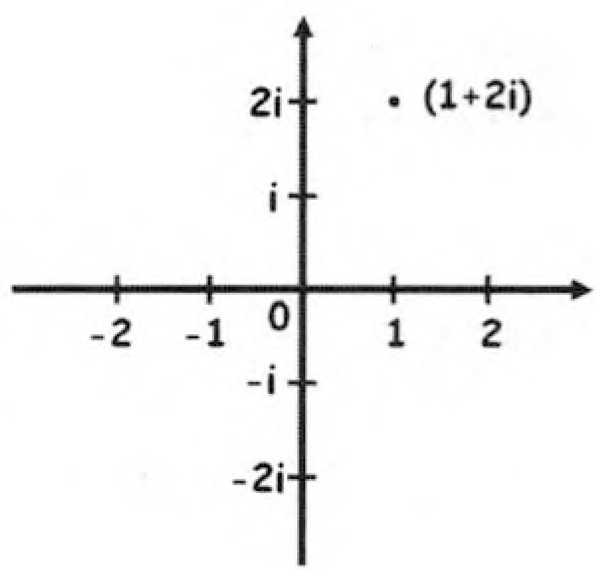


图14-10　复数在复数平面上的表示

介绍完了复数及复平面的基本知识后，就可介绍曼德布罗集了，其通过一个涉及复数的递归函数迭代产生，此递归函数如下。

$$Z_0 = 0 + 0i \quad \text{(初始条件)}$$

$$Z_{n+1} = Z_n^2 + c \quad \text{(迭代规则)}$$

从上述递归函数中可以看出，不同的常数c 会导致不同的迭代结果。有些c 值经过迭代可能会产生无穷大，有些可能不会。那些不会导致无穷大的c 值就构成了曼德布罗集。

可以通过OpenGL ES 2.0的片元着色器进行上述的迭代计算，将迭代一定次数后达到无穷大（实际开发中指超过指定值）的片元采用一种颜色着色，迭代一定次数后小于指定值的片元采用另一种颜色着色。如果希望得到更绚丽的图案，则可以将迭代一定次数后超过指定值的片元，根据迭代次数的多少采用不同的颜色着色，图14-11就给出了一幅采用不同灰度进行着色的曼德布罗集图案。

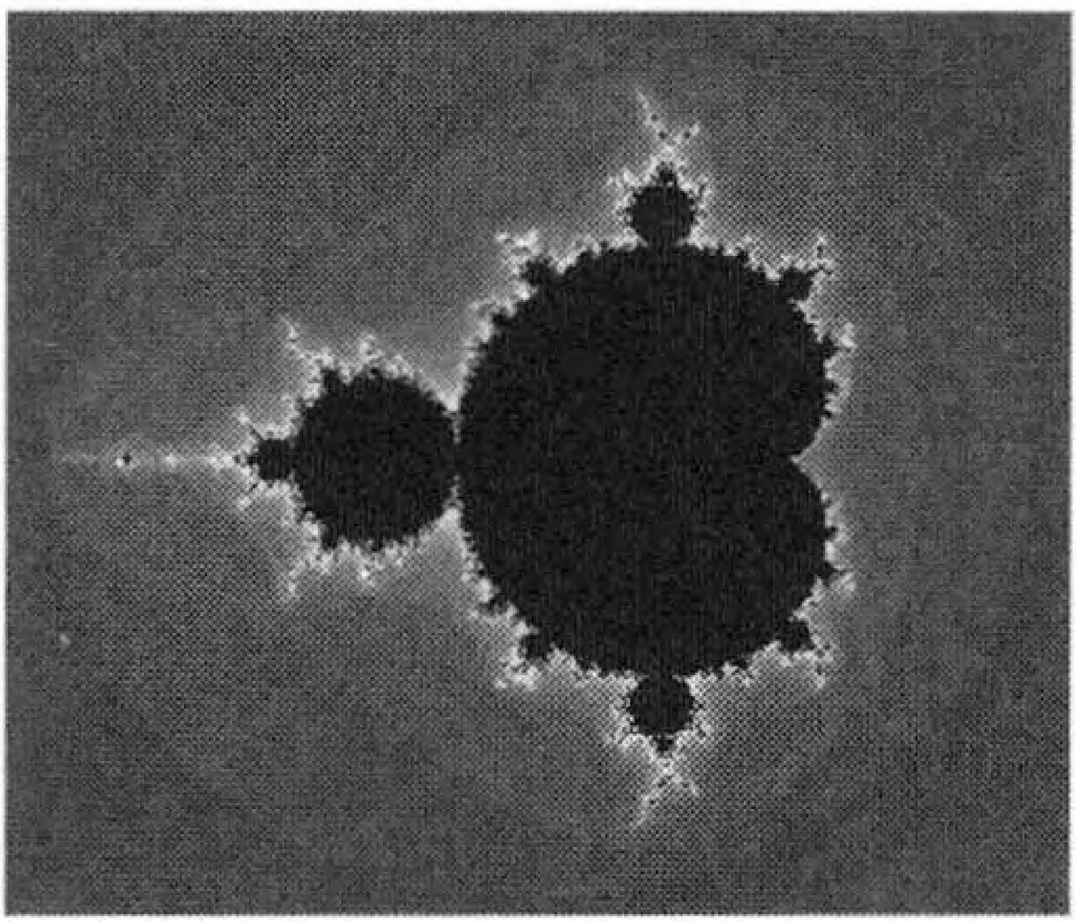


图14-11　曼德布罗物集合图形

14.3.2　曼德布罗集着色器的实现

上一小节介绍了曼德布罗集的基本原理，本小节将给出一个实现了曼德布罗集着色器的案例Sample14_7，其运行效果如图14-12所示。

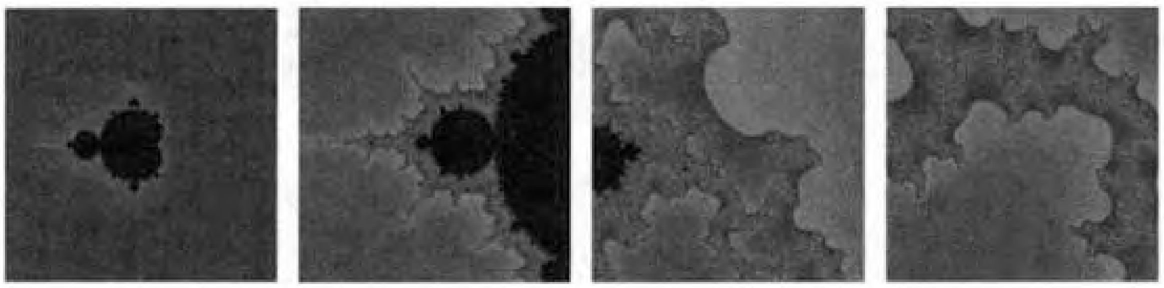


图14-12　案例Sample14_7的运行效果图

说明

图14-12中的4幅图是采用不同的中心坐标和缩放位置绘制的，4幅图各自的绘制参数如表14-5所列。学习完本案例的代码后，读者可以自由改变这些参数以获得想要的渲染效果。另外，由于本书插图采用的是灰度印刷，因此可能看起来不是很漂亮，此时请读者自行用真机运行本案例观察就可以看到很漂亮的曼德布罗集了。

表14-5　缩放系数和中心坐标位置的值

中心x坐标	中心y坐标	缩放系数
0	0	1
-1.36	0.005	0.1
-0.0002	0.7383	0.01
-0.0002	0.8383	0.001

了解了案例的运行效果后，下面对案例的开发进行简要的介绍。由于本案例中大部分代码与前面很多案例中的基本一致，因此这里仅介绍本案例中有代表性的着色器部分，具体内容如下。

（1）首先给出的是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第14章/Sample14_7/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                               //总变换矩阵
attribute vec3 aPosition;                              //顶点位置
attribute vec2 aTexCoor;                               //顶点纹理坐标
varying vec2 vTexPosition;                             //转换后传递给片元着色器的纹理坐标
void main() {
   gl_Position = uMVPMatrix * vec4(aPosition,1);  //根据总变换矩阵计算此次绘制此顶点的位置
   vTexPosition = (aTexCoor-0.5)*5.0;                  //将纹理坐标转换后传递给片元着色器
}
```

提示

从上述代码中可以看出，此顶点着色器与普通纹理映射的顶点着色器基本一致。唯一的区别就是其不是将管线传入的纹理坐标直接传出的，而是将纹理坐标从0.0~1.0的范围转换到从-2.5~2.5的范围后再传出的，这样做是为了后面片元着色器迭代计算的方便。

（2）接着给出的是片元着色器，其代码如下。

代码位置：见随书光盘中源代码/第14章/Sample14_7/assets目录下的frag.sh。

```
1 precision mediump float;                                    //给出默认的浮点精度
2 varying vec2 vTexPosition;                                  //从顶点着色器传递过来的纹理坐标
3 const float maxIterations = 9999.0;                         //最大迭代次数
4 const float zoom = 1.0;                                     //缩放系数
5 const float xCenter = 0.0;                                  //中心 X 坐标
6 const float yCenter = 0.0;                                  //中心 Y 坐标
7 const vec3 innerColor = vec3(0.0, 0.0, 1.0);                //内部颜色
8 const vec3 outerColor1 = vec3(0.1, 0.0, 0.0);               //外部颜色 1
9 const vec3 outerColor2 = vec3(0.0, 1.0, 0.0);               //外部颜色 2
10    void main() {
11        float real = vTexPosition.x * zoom + xCenter;       //变换当前位置
12        float imag = vTexPosition.y * zoom + yCenter;
13        float cReal = real;                                 //c 的实部
14        float cImag = imag;                                 //c 的虚部
15        float r2 = 0.0;                                     //半径的平方
16        float i;                                            //迭代次数
17        for(i=0.0; i<maxIterations && r2<4.0; i++){         //循环迭代
18            float tmpReal = real;                           //保存当前实部值
19            real = (tmpReal * tmpReal) - (imag * imag) +cReal;//计算下一次迭代后实部的值
20            imag = 2.0 *tmpReal * imag +cImag;              //计算下一次迭代后虚部的值
21            r2 = (real * real) + (imag * imag);             //计算半径的平方
22        }
23      vec3 color;                    //最终颜色
24      if(r2 < 4.0){                  //如果 r2 未达到 4 就退出了循环，表明迭代次数已达到最大值
25       color = innerColor;           //此时采用内部颜色对此片元着色
26      }else{                         //如果因 r2 大于 4.0 而退出循环，表明此位置在外部
27       color = mix(outerColor1, outerColor2, fract(i * 0.07));
                                       //按迭代次数采用不同的颜色着色
28        }
29        gl_FragColor = vec4(color, 1.0);   //将最终颜色传递给管线
30    }
```

• 第17-28行是本片元着色器的关键，其实现了上一小节中介绍的曼德布罗集迭代生成算法。首先通过一个循环进行迭代，当迭代超过最大次数或指定值时停止迭代。停止迭代后根据终止迭代的原因采用不同的颜色对片元进行着色。

• 通过调整缩放系数（zoom），中心点坐标（xCenter,yCenter）可以得到不同的局部图案，有兴趣的读者可以自行修改这些参数并运行观察。

提示

像曼德布罗集这样大剂量高并发的运算用片元着色器实现最好不过，性能将大大优于CPU的实现。笔者自己也开发过用Java代码基于CPU实现的版本（也是Android手机版的），需要数十秒才能跑出

结果，而本小节的案例仅需不到1秒。从这里也可以看出，在3D游戏开发中恰当运用计算能力较强的GPU可以很好地改善性能问题。

14.3.3　将曼德布罗集纹理应用到实际物体上

上一小节案例中的曼德布罗集纹理是应用到一个简单的纹理矩形上的，其实可以方便地将其应用到任意的物体上。本小节就给出一个将曼德布罗集纹理应用到茶壶上的案例Sample14_8，其运行效果如图14-13所示。



图14-13　案例Sample14_8的运行效果图

提示

由于本书插图采用的是灰度印刷，因此可能看起来不是很漂亮，此时请读者自行用真机运行本案例观察就可以看到很漂亮的曼德布罗集纹理茶壶了。

了解了案例的运行效果后，下面简要介绍一下案例的开发。由于本案例是复制并修改的第9章的案例Sample9_4，故没有变化的代码不再赘述，仅介绍着色器中有变化的部分，具体内容如下所列。

（1）首先需要将顶点着色器中直接将纹理坐标传入片元着色器的代码进行修改。修改为将纹理坐标从0.0～1.0的范围转换到从-2.5～2.5的范围后再传出的版本。这部分代码很简单，这里就不给出了，需要的读者请参考随书光盘里的源代码。

（2）接着将片元着色器进行修改，增加曼德布罗集迭代计算的相关代码。由于需要增加的代码与上一小节案例Sample14_7中的基本相同，因此这里不再重复给出，需要的读者也请参考随书光盘里的源代码。

提示

特别需要注意的是，本案例中迭代次数需要从上一小节的9999改为99，否则用手指在屏幕上滑动以旋转茶壶时就会很卡。

14.3.4 茱莉亚集着色器的实现

将曼德布罗集中与片元位置挂钩的常量c替换为固定常量后就可以产生茱莉亚集的分形图案，本小节的第一个案例Sample14_9就是实现了茱莉亚集分形图案的纹理矩形，其运行效果如图14-14所示。

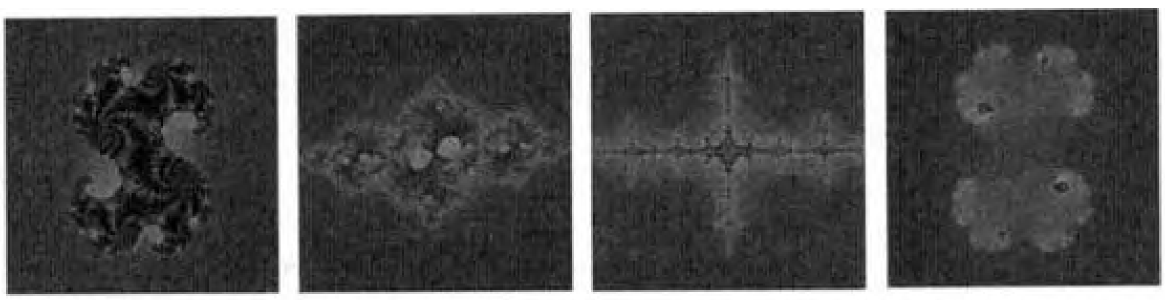


图14-14 Sample14_9运行效果图

说明

图14-14中的4幅图是采用不同的固定常量c绘制的，4幅图各自的绘制参数如表14-6所列。学习完本案例的代码后，读者可以自由改变这些参数以获得想要的渲染效果。另外，由于本书插图采用的是灰度印刷，因此可能看起来不是很漂亮，此时请读者自行用真机运行本案例观察就可以看到很漂亮的茱莉亚集了。

表14-6 不同的参数c

实部	虚部	实部	虚部
0.32	0.043	-0.765	0.11
-1.5	0.0	0.42	0.043

了解了案例的运行效果后，下面对案例的开发进行简要的介绍。由于本案例是由前面的案例Sample14_7修改而来，而且仅修改了片元着色器。故这里仅给出修改后片元着色器的代码，具体内容如下。

代码位置：见随书光盘中源代码/第14章/Sample14_9/assets目录下的frag.sh。

```
1 precision mediump float;                                   //给出默认的浮点精度
2 varying vec2 vTexPosition;                                 //从顶点着色器传递过来的纹理坐标
3 const float maxIterations = 9999.0;                        //最大迭代次数
4 const float zoom = 0.6;                                    //缩放系数
5 const float xCenter = 0.0;                                 //中心X坐标
6 const float yCenter = 0.0;                                 //中心Y坐标
7 const vec3 innerColor = vec3(0.0, 0.0, 1.0);               //内部颜色
8 const vec3 outerColor1 = vec3(0.1, 0.0, 0.0);              //外部颜色1
9 const vec3 outerColor2 = vec3(0.0, 1.0, 0.0);              //外部颜色2
10    void main() {
11        float real = vTexPosition.x * zoom + xCenter;      //变换当前位置
12        float imag = vTexPosition.y * zoom + yCenter;
13        float cReal = 0.32;                                    //c的实部
```

```
14        float cImag =0.043;                                    //c的虚部
15        float r2 = 0.0;                                        //半径的平方
16        float i;                                               //迭代次数
17        for(i=0.0; i<maxIterations && r2<4.0; i++){            //循环迭代
18            float tmpReal = real;                              //保存当前实部值
19            real = (tmpReal * tmpReal) - (imag * imag) +cReal;//计算下一次迭代后实部的值
20            imag = 2.0 *tmpReal * imag +cImag;                 //计算下一次迭代后虚部的值
21            r2 = (real * real) + (imag * imag);                //计算半径的平方
22        }
23       vec3 color;                  //最终颜色
24       if(r2 < 4.0){                //如果r2未达到4就退出了循环，表明迭代次数已达到最大值
25        color = innerColor;         //此时采用内部颜色对此片元着色
26       }else{                       //如果因r2大于4.0而退出循环，表明此位置在外部
27        color = mix(outerColor1, outerColor2, fract(i * 0.07));
                                      //按迭代次数采用不同的颜色着色
28       }
29        gl_FragColor = vec4(color, 1.0);                       //将最终颜色传递给管线
30    }
```

提示

从上述代码中可以看出，最大的变化就是第13行与14行常数c的实部与虚部。修改前实部与虚部是与位置挂钩的，修改后变成固定的常量了。

同样也可以将茱莉亚集分形纹理应用到茶壶上，只要将前面的案例Sample14_8复制一份并对片元着色器进行相应的修改即可得到案例Sample14_10，其运行效果如图14-15所示。



图14-15　Sample14_10运行效果图

提示

由于本书插图采用的是灰度印刷，因此可能看起来不是很漂亮，此时请读者自行用真机运行本案例观察就可以看到很漂亮的茱莉亚集纹理茶壶了。

由于本案例中的代码主要是来自案例Sample14_8，仅仅是将原来曼德布罗集的片元着色器修改成了茱莉亚集的片元着色器。而茱莉亚集片元着色器的代码前面也已经给出，故这里就不再给出本案例的代码了，需要的读者请参考随书光盘。

14.4　本章小结

本章主要介绍了片元着色器的几种巧妙用法，包括程序纹理技术、数字图像处理技术及分形集合的可视化等。通过本章的学习，读者可以初窥片元着色器在开发中的巨大魅力，为以后开发出具有吸引力的应用程序打下坚实的基础。

第15章 真实光学环境的模拟

本章将在前面章节知识的基础上进一步向读者介绍如何通过OpenGL ES 2.0模拟现实世界的一些常见光学效果，如投影、反射、折射、阴影等。掌握了这些技术以后，读者将能开发出更加真实、更加吸引用户的酷炫场景。

15.1　投影贴图

现实世界中很多场合都会有如下情况：在光源前面放一张透明胶片，胶片上面有半透明的图案，经光源照射后透明胶片上的图案将投

射到被光源照射的物体上。本节将通过一个案例Sample15_1向读者介绍如何基于OpenGL ES 2.0来开发这样的场景。

15.1.1　案例效果与基本原理

介绍本案例的具体开发步骤之前首先需要了解一下案例的运行效果与基本原理，其运行效果如图15-1所示。

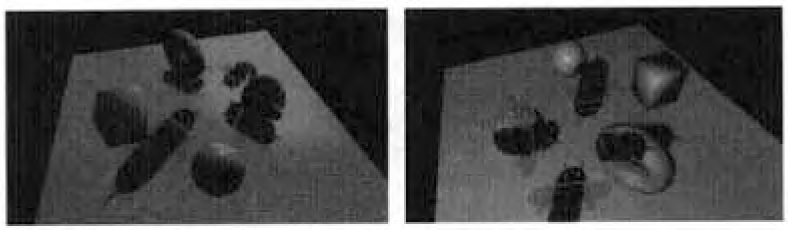


图15-1　投影贴图案例运行效果图

从运行效果图中可以看出，光源位于场景的一侧（靠近圆环与茶壶，距离立方体与圆球较远），从光源处将纹理图（如图15-2所示）投射到整个场景中。而且被光源投射的纹理图还编程实现了旋转的效果，故场景中的投影是不断变化的。

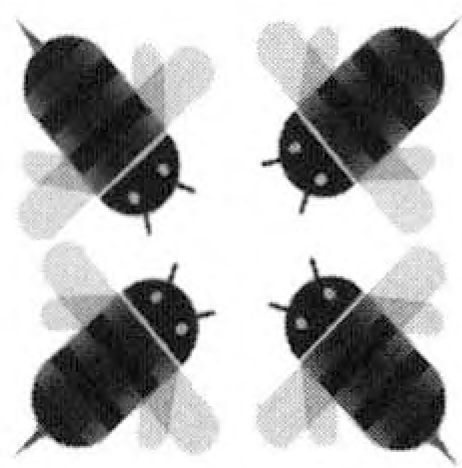


图15-2　投影用纹理

提示

运行本案例时当手指在屏幕上水平滑动时摄像机会绕场景转动，当手指在屏幕上垂直滑动时摄像机会随手指的移动而升降。

实现上述场景的基本技术是纹理映射（这种特殊的纹理映射也被称为投影贴图），只不过与前面章节介绍的普通纹理映射不同，这里的纹理坐标不是预先在初始化时分配给顶点的，而是在着色 器中根据传入的相关参数实时计算出来的，基本思想如图15-3所示。

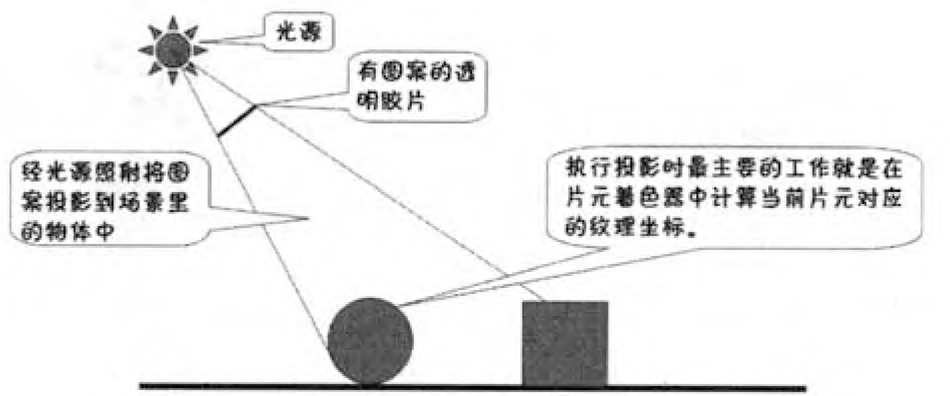


图15-3　投影贴图的基本原理

从图15-3中可以看出，此案例中最重要的工作就是在着色器中根据光源位置、透明胶片纹理图的位置、尺寸及片元的位置计算出片元对应的纹理坐标。这项工作如果直接用空间解析几何计算虽然是可以完成的，但会非常烦琐。

其实转变一下思维方式就很容易实现，可以在光源处虚拟一个摄像机，对应于此虚拟摄像机的投影参数进行如下设置。

- 将left与right各设置为0.5，总和为1，代表透明胶片纹理图的宽度纹理坐标跨度。
- 将top与bottom也各自设置为0.5，总和为1，代表透明胶片纹理图的高度纹理坐标跨度。
- 将near设置为透明胶片距光源的距离，far设置为不小于光源到需要照射的最远物体的距离。然后将摄像机观察矩阵及投影矩阵的组合矩阵传入着色器，在着色器中将片元位置与此矩阵进行运算即可计算出此片元位置投影到光源处虚拟摄像机近平面上的位置。由于前面将近平面的尺寸设置为1.0×1.0了，所以此近平面上的位置可以非常方便地换算成合法的纹理坐标，如图15-4所示。

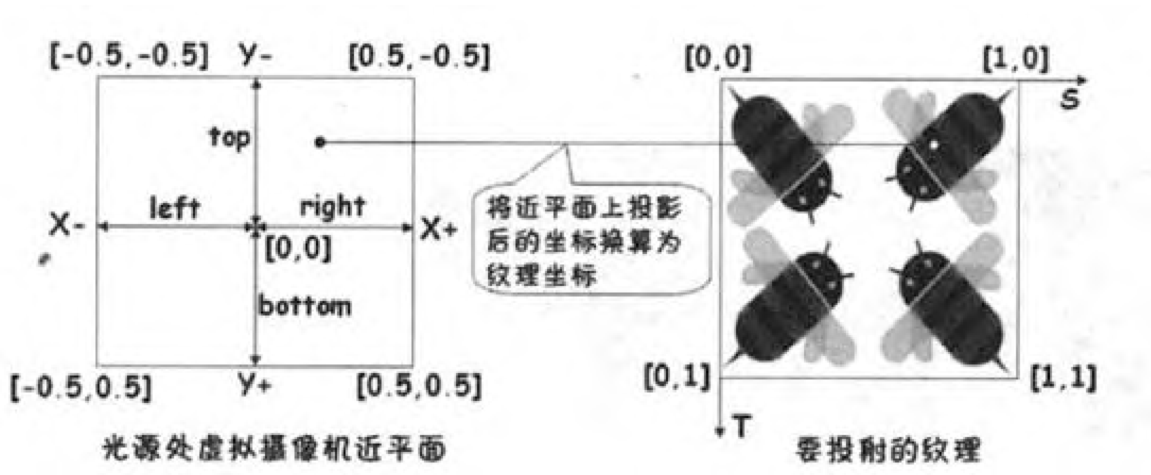


图15-4　纹理坐标换算

从图中可以看出换算公式为。

s=x+0.5　t=y+0.5

计算出纹理坐标后就非常简单了，只需要将纹理坐标传递给纹理采样函数进行纹理采样即可得到片元的颜色，投影贴图也就实现了。

15.1.2　开发步骤

了解了案例的运行效果及基本原理后，就可以进行案例的开发了，具体步骤如下。

（1）首先用3DMax生成5个基本物体（平面、圆环、茶壶、立方体、圆球），并导出成obj文件放入项目的assets目录待用。

（2）开发出搭建场景的基本代码，包括：加载物体、摆放物体、计算光照等。这些代码与前面章节的许多案例基本套路完全一致，因此这里不再赘述。

（3）由于需要将光源处虚拟摄像机的投影与观察组合矩阵传入着色器，因此需要对MatrixState类进行升级，增加获取投影与观察组合矩阵的getViewProjMatrix方法，其代码如下。



代码位置：见随书光盘中源代码/第15章/Sample15_1/com/bn/Sample15_1目录下的MatrixState.java。

```
public static float[] getViewProjMatrix() {
 float[] mMVPMatrix=new float[16];      //创建结果矩阵数组
 Matrix.multiplyMM(mMVPMatrix, 0, mProjMatrix, 0, mVMatrix, 0);
                                        //将投影及观察矩阵相乘得到组合矩阵
 return mMVPMatrix;                     //返回组合矩阵
}
```

（4）在MySurfaceView中绘制场景时需要比原来增加获取投影与观察组合矩阵的代码，具体情况如下。

代码位置：见随书光盘中源代码/第15章/Sample15_1/com/bn/Sample15_1目录下的MySurfaceView.java。

```
public void onDrawFrame(GL10 gl) {
MatrixState.setCamera(lx,ly,lz,10f,0f,10f,ux,uy,uz); //产生位于光源处虚拟摄像机的观察矩阵
MatrixState.setProjectFrustum(-0.5f, 0.5f, -0.5f, 0.5f, 1f, 400);
                                            //产生位于光源处虚拟摄像机的投影矩阵
mMVPMatrixGY=MatrixState.getViewProjMatrix();   //获取虚拟摄像机的观察、投影组合矩阵
GLES20.glClear(                             //清除深度缓冲与颜色缓冲
 GLES20.GL_DEPTH_BUFFER_BIT | GLES20.GL_COLOR_BUFFER_BIT);
MatrixState.setCamera(cx,cy,cz,0f,0f,0f,0f,1f,0f); //生成实际摄像机观察矩阵
MatrixState.setProjectFrustum(-ratio, ratio, -1.0f, 1.0f, 2, 1000);//生成实际摄像机投影矩阵
MatrixState.setLightLocation(lx, ly, lz);       //设置光源位置
    ……//此处省略绘制场景中各个物体的代码，需要的读者请参考随书光盘
}
```

（5）接着还需要在MySurfaceView类构造器中增加定时修改光源处虚拟摄像机对应up向量的代码，具体内容如下。

代码位置：见随书光盘中源代码/第15章/Sample15_1/com/bn/Sample15_1目录下的MySurfaceView.java。

```
new Thread() {
 public void run(){
  while(true){
     lAngle += 0.5;                              //改变up向量绕Y轴的旋转角度
     ux=(float) Math.sin(Math.toRadians(lAngle))*lR;//根据角度计算当前up向量的X分量
     uz=(float) Math.cos(Math.toRadians(lAngle))*lR; //根据角度计算当前up向量的Z分量
     try { Thread.sleep(20); } catch (InterruptedException e) {e.printStackTrace();}
                                                 //线程休眠
} } }.start();
```

说明

上述代码中的线程定时（每20ms）修改光源处虚拟摄像机对应up向量绕Y轴旋转的角度，以在程序中实现被投影的纹理图不断旋转的效果。

（6）绘制加载物体的类也要相对于前面章节的版本进行升级，增加传递位于光源处虚拟摄像机投影、观察组合矩阵进渲染管线的相关代码。这部分代码与传递其他矩阵的代码套路相同，这里不再赘述，需要的读者请参考随书光盘。

（7）本案例中的顶点着色器代码与前面光照部分的基本相同，仅仅增加了将顶点位置传递给片元着色器的功能，这里也不再赘述，需要的读者请参考随书光盘。

（8）本案例中的片元着色器是实现投影贴图的关键所在，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_1/assets目录下的frag.sh。

```
1 precision mediump float;          //设置默认精度
2 uniform sampler2D sTexture;       //纹理内容数据
3 varying vec4 ambient;             //接收从顶点着色器传递过来的环境光参数
4 varying vec4 diffuse;             //接收从顶点着色器传递过来的散射光参数
5 varying vec4 specular;            //接收从顶点着色器传递过来的镜面光参数
6 varying vec4 vPosition;           //接收从顶点着色器传递过来的片元位置
7 uniform highp mat4 uMVPMatrixGY;  //光源位置处虚拟摄像机观察及投影组合矩阵
8 void main() {
9       //将片元的位置投影到光源处虚拟摄像机的近平面上
10      vec4 gytyPosition=uMVPMatrixGY * vec4(vPosition.xyz,1);
11      gytyPosition=gytyPosition/gytyPosition.w;//进行透视除法
12      float s=gytyPosition.s+0.5;              //将投影后的坐标换算为纹理坐标
13      float t=gytyPosition.t+0.5;
14      vec4 finalColor=vec4(0.8,0.8,0.8,1.0);   //物体本身的颜色
15      if(s>=0.0&&s<=1.0&&t>=0.0&&t<=1.0){      //若纹理坐标在合法范围内则考虑投影贴图
16            vec4 projColor=texture2D(sTexture, vec2(s,t)); //对投影纹理图进行采样
17            vec4 specularTemp=projColor*specular;      //计算投影贴图对镜面光的影响
18            vec4 diffuseTemp=projColor*diffuse;        //计算投影贴图对散射光的影响
19            //计算最终片元颜色
20            gl_FragColor = finalColor*ambient+finalColor*specularTemp+finalColor
              *diffuseTemp;
21      } else {//计算最终片元颜色
22          gl_FragColor = finalColor*ambient+finalColor*specular+finalColor*diffuse;
23      }
24   }
```

- 第10-13行完成了最核心的工作，首先将此片元投影到位于光源位置的摄像机对应的近平面上，然后将投影坐标换算为纹理坐标。

• 第16-23行首先判断换算出的纹理坐标是否在合理的范围内（0.0～1.0），若在则对投影纹理图进行采样，并将采样结果与散射光、镜面光分量进行加权计算，最终将3个光通道与物体颜色加权计算得到片元颜色。若纹理坐标不在合理范围内，则直接进行3个光通道与物体颜色的加权计算。

到这里为止投影贴图技术就介绍完了，在以后的具体项目开发中读者若能做到灵活运用，将可以开发出更加逼真的场景。

15.2　反射环境模拟

现实世界中很多场合都会有这样的情况，放置在环境中具有良好反射表面的物体会反射出周围环境的内容。例如，在环境中放置了一套表面抛光的银质餐具，餐具的光滑表面就会映射出周围环境的内容。本节将通过一个案例Sample15_2向读者介绍如何基于OpenGL ES 2.0来开发这样的效果。

15.2.1　案例效果与基本原理

介绍本案例的具体开发步骤之前首先需要了解一下案例的运行效果与基本原理，其运行效果如图15-5所示。



图15-5　反射环境纹理案例运行效果图

从运行效果图中可以看出，本节案例在一个自然场景中放置了一把具有光滑反射表面的茶壶，茶壶表面反射出了周围场景中的内容。自然场景是用前面章节介绍过的天空盒技术实现的，光滑表面的茶壶反射周围的环境内容是通过立方图纹理技术实现的。

提示

本案例运行时，当手指在屏幕上水平滑动时摄像机会绕场景转动，当手指在屏幕上垂直滑动时摄像机会随手指的移动而升降。

立方图纹理是一种特殊的纹理映射技术，主要包括如下两个要点。

- 立方图纹理的单位是套，一套立方图纹理包括6幅尺寸相同的正方形纹理图。与构造天空盒的思路相同，这6幅图正好包含了周天360°全部的场景内容。
- 对立方图纹理进行采样时，需要给出的不再是S、T两个轴的纹理坐标，而是一个规格化的向量。此规格化向量代表采样的方向，用来确定在代表全周天360°的6幅图中的哪一幅的哪个位置进行采样，具体情况如图15-6所示。

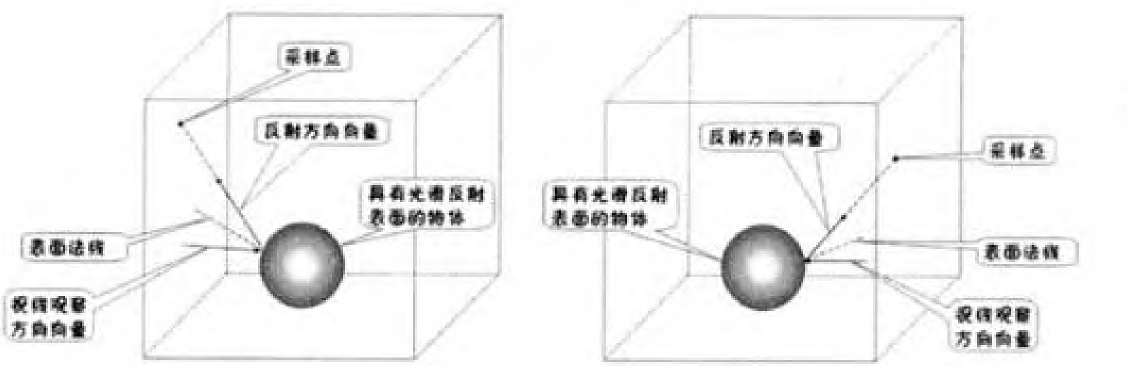


图15-6　立方图纹理采样技术实现反射效果的原理

从图15-6中可以看出，本案例中实现茶壶反射周围环境的内容有两项关键工作，具体如下所列。

- 根据摄像机位置及被观察点位置计算出观察方向（视线）向量，并参照被观察点的法向量计算出视线反射方向向量。
- 根据视线反射方向向量确定采样点，实施立方图纹理采样。这与现实世界中人眼观察光滑物体，其表面有反射是完全一致的。

提示

上述两项工作中的第一项需要自己编程在着色器中实现，第二项只需要拿着采样向量调用textureCube函数即可完成。

15.2.2　开发步骤

了解了案例的运行效果及基本原理后，就可以进行案例的开发了，具体步骤如下。

（1）首先用3DMax生成一个茶壶，并导出成obj文件放入项目的assets目录中待用。

（2）开发出搭建场景的基本代码，包括：天空盒、加载物体、摆放物体等。这些代码与前面章节的许多案例基本套路完全一致，因此这里不再赘述。

（3）由于本案例中茶壶采用的是立方图纹理，所以在MySurfaceView类中需要首先增加初始化立方图纹理的工具方法generateCubeMap，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_2/com/bn/Sample15_2目录下的MySurfaceView.java。

```
1 public int generateCubeMap(int[] resourceIds) {
2   int[] ids = new int[1];                    //用于存储生成的纹理编号的数组
3   GLES20.glGenTextures(1, ids, 0);           //生成一个新的纹理编号
4   int cubeMapTextureId = ids[0];
5   GLES20.glBindTexture(GLES20.GL_TEXTURE_CUBE_MAP, cubeMapTextureId);//绑定纹理
6   //设置纹理采样方式、拉伸方式
7   GLES20.glTexParameterf(GLES20.GL_TEXTURE_CUBE_MAP,
8       GLES20.GL_TEXTURE_MIN_FILTER, GLES20.GL_LINEAR);
9   GLES20.glTexParameterf(GLES20.GL_TEXTURE_CUBE_MAP,
10          GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_LINEAR);
11      GLES20.glTexParameterf(GLES20.GL_TEXTURE_CUBE_MAP,
12          GLES20.GL_TEXTURE_WRAP_S,GLES20.GL_REPEAT);
13      GLES20.glTexParameterf(GLES20.GL_TEXTURE_CUBE_MAP,
14          GLES20.GL_TEXTURE_WRAP_T,GLES20.GL_REPEAT);
15      for (int face = 0; face < 6; face++){//循环加载立方图纹理各个面的图片
16        InputStream is = getResources().openRawResource(resourceIds[face]);
17        Bitmap bitmap;
18        try { bitmap = BitmapFactory.decodeStream(is); } finally {
19          try {is.close();} catch(IOException e) {
20              Log.e("CubeMap", "Could not decode texture for face " + Integer.toString
                (face));}}
21        GLUtils.texImage2D(  //加载立方图纹理中的指定编号的面
22          GLES20.GL_TEXTURE_CUBE_MAP_POSITIVE_X + face, 0,bitmap, 0);
23        bitmap.recycle();      //释放图片所占内存
24      }
25      return cubeMapTextureId;     //返回加载完成的立方图纹理编号
26    }
```

• 第1行方法的入口参数与以前加载纹理的方法不同，不再是一幅图片的资源id，而是一个资源id数组，其用来接收组成一套立方图纹理的6幅图片的资源id。

• 第7-14行设置了纹理的采样与拉伸方式，与前面的案例类似，但要注意的是所有设置方法的第一个参数不再是“GL_TEXTURE_2D”，而是“GL_TEXTURE_CUBE_MAP”。

• 第15-24行循环将立方图纹理中各个面的图片加载进内存，然后再加载进显存。

（4）开发完了加载立方图纹理的generateCubeMap方法后，就可以在MySurfaceView类的onSurfaceCreated方法中调用其进行立方图纹理的初始化了，代码片段如下。

代码位置：见随书光盘中源代码/第15章/Sample15_2/com/bn/Sample15_2目录下的MySurfaceView.java。

```
int[] cubeMapResourceIds = new int[]{  //组织图片资源 id 数组
  R.raw.skycubemap_right, R.raw.skycubemap_left, R.raw.skycubemap_up_cube,
  R.raw.skycubemap_down, R.raw.skycubemap_front, R.raw.skycubemap_back};
textureIdCM=generateCubeMap(cubeMapResourceIds); //加载立方图纹理
```

提示

请读者注意图片资源id的顺序为“右、左、上、下、前、后”，如果顺序不对多幅纹理图可能会拼接不上。

（5）完成了上述工作后只要在负责绘制茶壶的LoadedObjectVertexNormalTexture类中的drawSelf方法中增加绑定立方图纹理的相关代码即可，具体内容如下。

代码位置：见随书光盘中源代码/第15章/Sample15_2/com/bn/Sample15_2目录下的LoadedObjectVertexNormalTexture.javao

```
GLES20.glEnable(GLES20.GL_TEXTURE_CUBE_MAP);             //启用立方图纹理
GLES20.glBindTexture(GLES20.GL_TEXTURE_CUBE_MAP,texId);//绑定立方图纹理
```

提示

LoadedObjectVertexNormalTexture类中的其他代码与前面案例中的基本没有区别，需要的读者请自行参考随书光盘中的源代码。

（6）完成了Java代码之后就可以开发着色器了。本案例中有两套着色器：一套是普通贴纹理的，用来绘制天空盒的各个面；另一套是负责茶壶立方图纹理实施的。第一套与原来案例中的完全一样，这里不再赘述。下面着重介绍用来绘制茶壶的、立方图纹理相关的着色器，首先是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_2/assets目录下的vertex_tex_cube.sh。

```
uniform mat4 uMVPMatrix;   //总变换矩阵
uniform mat4 uMMatrix;         //变换矩阵
uniform vec3 uCamera;      //摄像机位置
attribute vec3 aPosition;      //顶点位置
attribute vec3 aNormal;        //顶点法向量
varying vec3 vTextureCoord;  //用于传递给片元着色器的立方图采样向量
void main() {
  gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点位置
  //计算变换后的法向量并规格化
```

```
      vec3 normalTarget=aPosition+aNormal;
      vec3 newNormal=(uMMatrix*vec4(normalTarget,1)).xyz-(uMMatrix*vec4(aPosition,1)).xyz;
      newNormal=normalize(newNormal);
      //计算从观察点到摄像机的向量(视线向量)
      vec3 eye=- normalize(uCamera-(uMMatrix*vec4(aPosition,1)).xyz);
      vTextureCoord=reflect(-eye,newNormal);     //计算视线向量的反射向量并传递给片元着色器
  }
```

说明

上述顶点着色器与实现普通纹理映射功能的顶点着色器最大的区别是，传递给片元着色器的纹理坐标不是从管线接收并直接赋值给易变变量的二维向量，而是通过视线及法向量计算出来的三维向量，代表视线经表面反射后的方向。

（7）完成了立方图纹理相关顶点着色器的开发后，就可以开发对应的片元着色器了，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_2/assets目录下的frag_tex_cube.sh。

```
precision mediump float;
uniform samplerCube sTexture;  //纹理内容数据
varying vec3 vTextureCoord;    //接收从顶点着色器过来的参数
void main() {
  //通过传入的采样向量与立方图纹理调用 textureCube 方法执行采样
  gl_FragColor=textureCube(sTexture, vTextureCoord);
}
```

> **说明**
>
> 从上述代码中可以看出，由于OpenGL ES 2.0着色语言内置函数的良好支持，得到采样向量后的采样工作就非常简单了，直接调用textureCube函数即可完成。

到这里为止如何采用立方图纹理技术实现反射环境模拟就介绍完了，在以后的具体项目开发中读者若能做到灵活运用，将可以开发出更加逼真的场景。

15.3 折射环境模拟

现实世界中很多场合都会有这样的情况，透过透明物体观察周围的环境时会有折射的现象。平时使用的很多光学设备利用的就是光线的折射，如：放大镜、眼镜、望远镜、显微镜等。本节将通过一个案例Sample15_3向读者介绍如何基于OpenGL ES 2.0来开发折射的效果。

15.3.1 案例效果与基本原理

介绍本案例的具体开发步骤之前首先需要了解一下案例的运行效果与基本原理，其运行效果如图15-7所示。



图15-7 反射环境纹理案例运行效果图

从运行效果图中可以看出，本节案例在自然场景中放置了一个透明的玻璃球，透过玻璃球观察后面的场景时会有放大镜的效果。自然

场景是用前面章节介绍过的天空盒技术实现的，玻璃球折射出周围环境的内容是通过立方图纹理技术实现的。

提示

本案例运行时，当手指在屏幕上水平滑动时摄像机会绕场景转动，当手指在屏幕上垂直滑动时摄像机会随手指的移动而升降。

本节案例与上一节反射的案例均采用立方体纹理技术，所不同的是本节中的采样向量是视线经玻璃球表面折射后的向量，而不再是被表面反射后的向量。立方体纹理技术本身已经在上一节详细介绍过，因此本节仅介绍采用立方图纹理技术实现折射效果的原理，如图15-8所示。

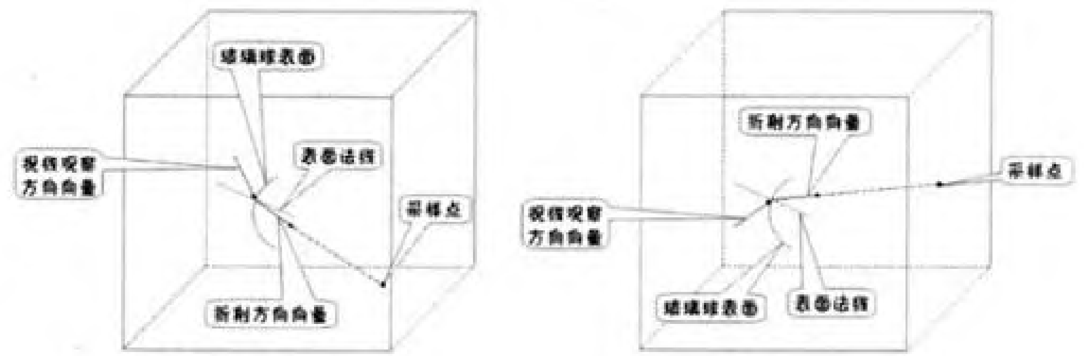


图15-8　立方体纹理技术实现折射效果的原理

从图15-8中可以看出，本案例中为了实现玻璃球折射出周围环境的内容也有两项关键工作要做，具体如下所列。

- 根据摄像机位置及被观察点位置计算出观察方向（视线）向量，并参照被观察点的法向量计算出视线折射方向向量。
- 根据视线折射方向向量确定采样点，实施立方图纹理采样。这与现实世界中人眼观察透明物体，其折射出观察的环境内容是完全一致的。

提示

上述两项工作中的第一项需要自己编程在着色器中实现，第二项只需要拿着采样向量调用textureCube函数即可完成。

折射方向的计算遵循斯涅尔定律，具体情况如图15-9所示。

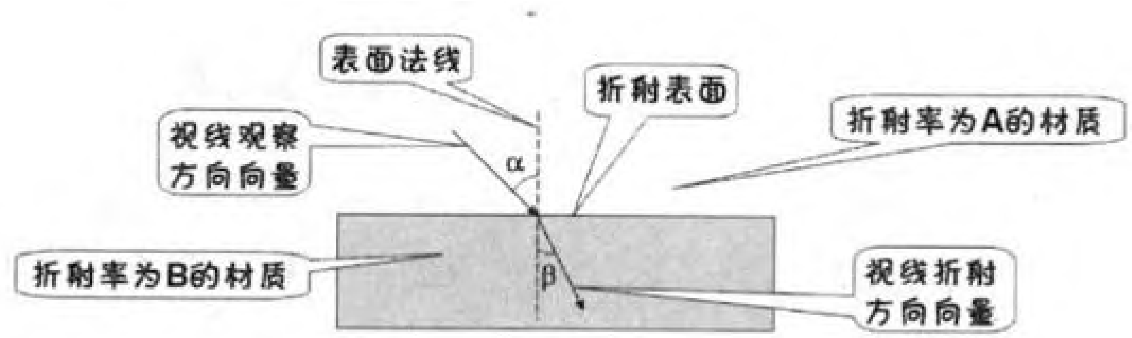


图15-9　斯涅尔定律原理

从图15-9中可以看出，入射向量与折射向量共面，且入射角α、折射角β与两种材质的折射率满足如下公式。

Axsin(α)=Bxsin(β)

提示

了解了折射的基本原理与计算公式后，下面就可以进行实际的开发了，下一小节将对这些内容进行介绍。

15.3.2　开发步骤

了解了案例的运行效果及基本原理后，就可以进行案例的开发了，具体步骤如下。

（1）首先用3DMax生成一个球，并导出成obj文件放入项目的assets目录中待用。

（2）开发出搭建场景的基本代码，包括：天空盒、立方图纹理、加载物体、摆放物体等。这些代码与前面章节的许多案例基本套路完全一致，因此这里不再赘述。

（3）完成了Java代码之后就可以开发着色器了。本案例中有两套着色器：一套是普通贴纹理的，用来绘制天空盒的各个面；另一套是

负责玻璃球立方图纹理实施的。第一套与原来案例中的完全一样，这里不再赘述。下面着重介绍用来绘制玻璃球折射效果的、立方图纹理相关的着色器，首先是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_3/assets目录下的vertex_tex_cube.sh。

```
uniform mat4 uMVPMatrix;          //总变换矩阵
uniform mat4 uMMatrix;            //变换矩阵
uniform vec3 uCamera;             //摄像机位置
attribute vec3 aPosition;         //顶点位置
attribute vec3 aNormal;           //顶点法向量
varying vec3 eyeVary;             //用于传递给片元着色器的视线向量
varying vec3 newNormalVary;       //用于传递给片元着色器的变换后法向量
void main()  {
  gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点位置
      //计算变换后的法向量并规格化
     vec3 normalTarget=aPosition+aNormal;
     vec3 newNormal=(uMMatrix*vec4(normalTarget,1)).xyz-(uMMatrix*vec4(aPosition,1)).xyz;
     newNormal=normalize(newNormal);
     newNormalVary=newNormal;//将变换后的法向量传递给片元着色器
     //计算从观察点到摄像机的向量(视线向量)
     vec3 eye = normalize(uCamera-(uMMatrix*vec4(aPosition,1)).xyz);
     eyeVary=eye;//将视线向量传递给片元着色器
   }
```

> **说明**
>
> 上述顶点着色器的主要功能为将视线向量、法向量通过易变变量传递给片元着色器，供片元着色器在折射计算中使用。

（4）完成了顶点着色器的开发后，就可以开发对应的片元着色器了，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_3/assets目录下的frag_tex_cube.sh。

```
1 precision mediump float;
2 uniform samplerCube sTexture;  //纹理内容数据
3 varying vec3 eyeVary;         //接收从顶点着色器过来的视线向量
4 varying vec3 newNormalVary;   //接收从顶点着色器过来的变换后法向量
5 vec4 zs(                      //根据法向量、视线向量及斯涅尔定律计算立方图纹理采样的方法
6   in float zsl                //折射系数
7 ){
8   vec3 vTextureCoord=refract(-eyeVary,newNormalVary,zsl);//根据斯涅尔定律计算折射后的视线向量
9   vec4 finalColor=textureCube(sTexture, vTextureCoord);   //进行立方图纹理采样
10      return finalColor;
11    }
12    void main(){
13      gl_FragColor=zs(0.94); //以折射系数0.94调用zs方法完成片元颜色的计算
14    }
```

- 第8行调用第4章介绍过的折射函数refract计算出折射后的视线向量。
- 第9行根据计算出的折射后视线向量调用textureCube函数进行立方图纹理采样。

> **提示**
>
> 从上述代码中可以看出，由于OpenGL ES 2.0着色语言内置函数的良好支持，很多工作都被大大地简化了。笔者强烈建议读者花些时间熟悉第4章介绍的函数，避免“重复发明车轮”，这样在开发中就可以事半功倍了。

到这里为止如何采用立方图纹理技术实现折射环境的模拟就介绍完了，在以后的具体项目开发中读者若能做到灵活运用，将可以开发出更加逼真的场景。

15.4　色散效果的模拟

上一小节通过玻璃球实现放大镜效果的案例介绍了折射效果的开发，读者应该觉得案例的整体效果还是非常逼真的。但还有一点不够真实的地方，那就是现实世界中不同颜色的光对于同一种材质的折射率是不完全相同的。因此，透过玻璃球观察现实世界时会产生色散的效果。本节将通过一个案例Sample15_4向读者介绍如何基于OpenGL ES 2.0来开发折射时色散的效果。

15.4.1 案例效果与基本原理

介绍本案例的具体开发步骤之前，首先需要了解一下案例的运行效果与基本原理，其运行效果如图15-10所示。



图15-10　折射色散案例运行效果图

从运行效果图中可以看出，本案例场景基本上与上一节的案例相同，唯一不同的就是在透过玻璃球观察后面的场景时不但会有放大镜的效果，物体的边缘还会有彩虹一样的条纹出现，这就是色散现象。

> **提示**
>
> 由于本书是黑白灰度印刷，因此通过上面的图15-10可能看不到色散的效果，这时请读者自行运行本节的案例或参看本书前面的彩页。

前面已经简单提过，色散的原因是不同颜色的光对于同一种材质的折射率不完全相同。这就是说一束由不同颜色的光组成的光线经过折射后，不同颜色的组成部分出射方向会有发散，形成彩虹一样的效果，图15-11很好地说明了这个问题。

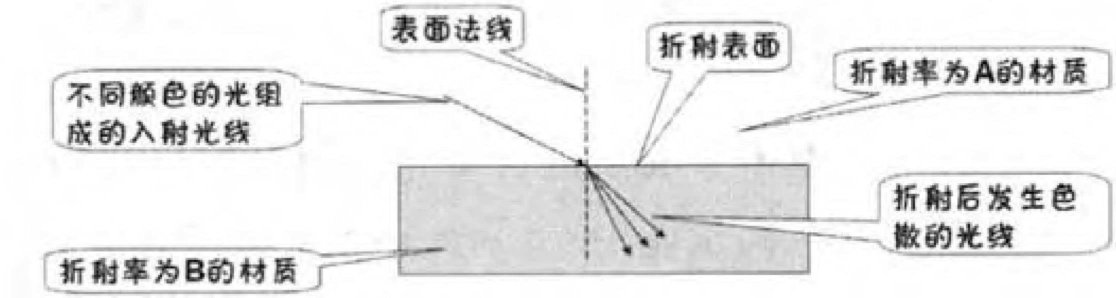


图15-11 折射色散的原理

> **提示**
>
> 了解了色散的基本原理与案例的运行效果后，下面就可以进行实际的开发了下一小节将对这些内容进行介绍。

15.4.2 开发步骤

由于本案例与上一节案例的大部分代码相同，仅仅片元着色器有所不同。因此这里仅给出改动后片元着色器的代码，具体内容如下。

代码位置：见随书光盘中源代码/第15章/Sample15_4/assets目录下的frag_tex_cube.sh。

```
1 precision mediump float;
2 uniform samplerCube sTexture;//纹理内容数据
3 varying vec3 eyeVary;          //接收从顶点着色器过来的视线向量
4 varying vec3 newNormalVary;    //接收从顶点着色器过来的变换后法向量
5 vec4 zs(                       //根据法向量、视线向量及斯涅尔定律计算立方图纹理采样的方法
6   in float zs1                 //折射率
7 ){
8   vec3 vTextureCoord=refract(-eyeVary,newNormalVary,zs1);//根据斯涅尔定律计算折射
9   vec4 finalColor=textureCube(sTexture, vTextureCoord);  //进行立方图纹理采样
10      return finalColor;
11    }
12    void main(){
13      vec4 finalColor=vec4(0.0,0.0,0.0,0.0);
14      //由于有色散 RGB 3 个色彩通道单独计算折射
15      finalColor.r=zs(0.97).r;     //计算红色通道的采样结果
16      finalColor.g=zs(0.955).g;    //计算绿色通道的采样结果
17      finalColor.b=zs(0.94).b;     //计算蓝色通道的采样结果
18      gl_FragColor=finalColor;   //将最终片元颜色传递给管线
19    }
```

提示

从上述代码中可以看出，大部分代码都没有变化，只是第15-17行将不同颜色的光分量以不同的折射系数单独进行了折射计算。

到这里为止如何实现折射色散效果的模拟就介绍完了，在以后的具体项目开发中读者若能做到灵活运用，将可以开发出更加逼真的场景。

15.5 菲涅尔效果的模拟

前面几节单独地介绍了反射、折射效果的模拟，本节将通过一个案例Sample15_5介绍将反射效果、折射效果综合到一起的菲涅尔（Fresnel）效果。

15.5.1 案例效果与基本原理

介绍本案例的具体开发步骤之前首先需要了解一下案例的运行效果与基本原理，其运行效果如图15-12所示。



图15-12　菲涅尔效果案例运行效果图

从运行效果图中可以看出，本案例场景基本上与前面一节的相同，所不同的就是场景中的玻璃球同时具有折射与反射效果。玻璃球正对摄像机的部分主要是折射效果，而边缘主要是反射效果，这就是前面提到的菲涅尔效果。

提示

由于本书是黑白灰度印刷，因此通过上面的图15-12可能看不清菲涅尔效果，这时请读者自行运行本节的案例或参看本书前面的彩页。

造成菲涅尔效果的原因是，当光线到达两种材质的接触面时，一部分光线被反射，另一部分光线被折射。大致的规律是当入射角较小时主要发生折射，当入射角较大时主要发生反射。

这与大家平时在湖边或池塘边的感觉一样：当目光相对于水面基本垂直时，主要看到的是水面下的内容；而当目光相对于水面入射角很大时，主要看到的是湖面反射的内容，而看不到水面下的内容，图15-13简单地说明了这个问题。

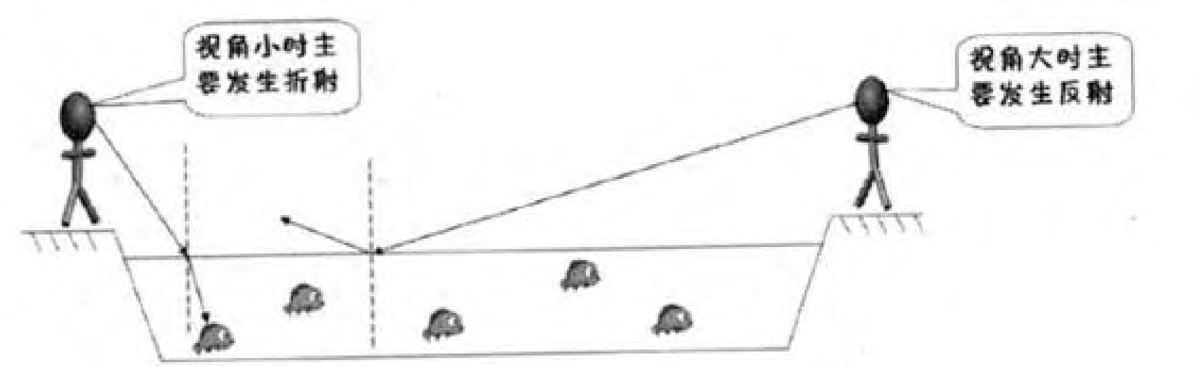


图15-13　菲涅尔效果

了解了菲涅尔效果的基本原理后还有一个重要的问题需要解决，那就是在给定情况下反射和折射各自所占的比例为多少。这个问题的精确计算比较复杂，需要用到专门为菲涅尔效果建立的复杂 数学模型，基于这些模型的计算难以满足游戏实时性的需要。

实际开发中笔者建议采用简化的数学模型，也就是将折反射比例分成3种情况进行计算，具体情况如下所列。

- 若入射角小于一定的值，只计算折射效果。
- 若入射角大于一定的值，只计算反射效果。
- 若入射角在一定的范围内，则首先单独计算折射效果与反射效果，再将两种效果的计算结果按一定的比例进行融合。

提示

了解了菲涅尔效果的基本原理与案例的运行效果后，下面就可以进行实际的开发了，下一小节将对这些内容进行介绍。

15.5.2　开发步骤

上一小节介绍了菲涅尔效果的基本原理及案例的运行效果，本小节将介绍如何进行本节案例的开发。由于本节案例与上一节案例的大部分代码相同，仅仅在片元着色器中有所不同。因此这里仅给出改动后片元着色器的代码，具体内容如下。

代码位置：见随书光盘中源代码/第15章/Sample15_5/assets目录下的frag_tex_cube.sh。

```
precision mediump float;
uniform samplerCube sTexture;  //纹理内容数据
varying vec3 eyeVary;          //接收从顶点着色器过来的视线向量
varying vec3 newNormalVary;    //接收从顶点着色器过来的变换后法向量
vec4 zfs( //计算折反射纹理采样颜色的方法
  in float zsl//折射率
){
  vec3 vTextureCoord;          //用于进行立方图纹理采样的向量
  vec4 finalColorZS;           //若是折射的采样结果
     vec4 finalColorFS;        //若是反射的采样结果
     vec4 finalColor;          //最终颜色
     const float maxH=0.7;     //入射角余弦值若大于此值则仅计算折射
     const float minH=0.2;     //入射角余弦值若小于此值则仅计算反射
     float sizeH=maxH-minH;//混合时余弦值的跨度
     float testValue=abs(dot(eyeVary,newNormalVary));   //计算视线向量与法向量的余弦值
     if(testValue>maxH)  {                              //余弦值大于 maxH 仅折射
       vTextureCoord=refract(-eyeVary,newNormalVary,zsl);
       finalColor=textureCube(sTexture, vTextureCoord);
     } else if(testValue<=maxH&&testValue>=minH) {//余弦值在 minH－maxH 范围内反射、折射融合
       vTextureCoord=reflect(-eyeVary,newNormalVary);
       finalColorFS=textureCube(sTexture, vTextureCoord);   //反射的计算结果
       vTextureCoord=refract(-eyeVary,newNormalVary,zsl);
       finalColorZS=textureCube(sTexture, vTextureCoord);   //折射的计算结果
       float ratio=(testValue-minH)/sizeH;          //融合比例
       finalColor=finalColorZS*ratio+(1.0-ratio)*finalColorFS;  //折反射结果线性融合
     } else  {                                      //余弦值小于 minH 仅反射
       vTextureCoord=reflect(-eyeVary,newNormalVary);
       finalColor=textureCube(sTexture, vTextureCoord);
     }
    return finalColor;                              //返回最终结果
  }
  void main(){
    vec4 finalColor=vec4(0.0,0.0,0.0,0.0);
    //由于有色散 RGB 3 个色彩通道单独计算折射
    finalColor.r=zfs(0.97).r;                       //计算红色通道
    finalColor.g=zfs(0.955).g;                      //计算绿色通道
    finalColor.b=zfs(0.94).b;                       //计算蓝色通道
    gl_FragColor=finalColor;                        //将最终的片元颜色传递给管线
  }
```

提示

从上述代码中可以看出，主要的变化在于将原来计算折射的ZS函数替换为综合计算折反射效果的zfs函数。此函数中根据入射角

度的大小分为3种情况计算纹理采样的结果，实现了上一小节提出的简化数学模型。

到这里为止如何实现菲涅尔效果的模拟就介绍完了，在以后的具体项目开发中读者若能做到灵活运用，将可以开发出更加逼真的场景。

15.6 凹凸映射

现实世界中有这样一类物体，其表面是粗糙的，具有很多微小细节。例如具有小花纹的茶壶、常吃的橘子等。当需要渲染这些物体时，可以采用前面章节介绍过的基于三角形的立体物体构建技术去描述这些微小细节。但这样势必造成顶点数量的急剧增加，大大降低渲染速度。

本节将介绍一种基于特殊纹理映射的低成本细节模拟技术——凹凸映射，其不需要增加很多的顶点来描述细节，只是在计算光照前通过运算扰动法向量，效率很高。

15.6.1 案例效果与基本原理

介绍凹凸映射的基本知识之前首先需要了解一下本节案例Sample15_6的运行效果，对凹凸映射有一个基本的认识，具体情况如图15-14所示。

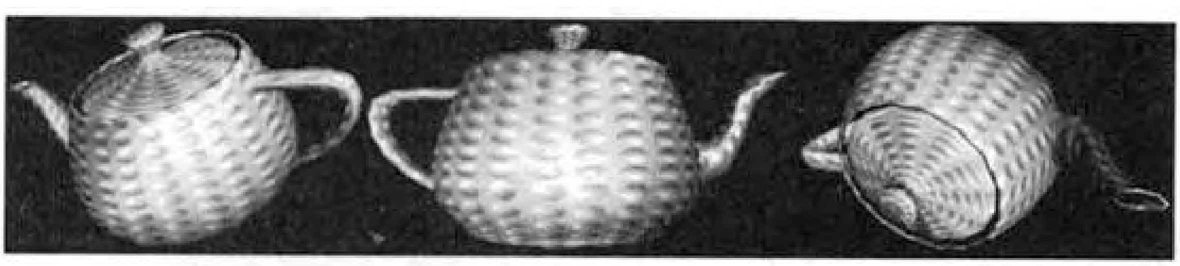


图15-14　凹凸映射案例的运行效果

从图15-14中可以看出，本案例中的茶壶表面具有很多椭圆形的小凸起，效果非常逼真。已经掌握了前面章节基于三角形构建立体物体

技术的读者可以很容易地分析出，若使用三角形来构建上述茶壶将需要数倍于普通茶壶的顶点数，这在当下的主流手机上是难以支持的。

幸运的是，Jim Blinn创造了凹凸映射技术，其使得仅仅需要采用普通茶壶的模型就可以绘制出具有微小细节的粗糙表面，此技术的基本思想如下所列。

- 首先绘制时采用的是没有表面细节的模型，并且模型中的每个顶点都指定了不考虑表面细节情况下的法向量、切向量以及一个纹理坐标。其中切向量指的是与顶点处表面相切与法向量正交的向量，在后面的计算中有重要作用。
- 给顶点分配的纹理坐标与前面章节中的案例一样也是用来进行纹理采样的，但被采样的纹理图中记录的不再是片元的颜色，而是此片元处考虑了微小细节的法向量相对于标准法向量的扰动结果。因此，每次纹理采样结果中的RGB通道值不再代表颜色，而分别代表扰动后法向量的X、Y、Z分量。
- 当取出扰动后法向量时，可以将其与此片元处不考虑表面细节的法向量进行计算得出此片元处考虑了表面细节后的法向量。最后用计算后的法向量进行正常的光照计算即可渲染出具有表面粗糙细节的物体。

一般情况下为了计算方便，纹理图中记录扰动结果法向量时，认为扰动前的标准法向量为[0，0，1]。也就是说从纹理图中取出的扰动后法向量是以标准法向量[0，0，1]为基础的，而并不是基于对应片元处不考虑表面细节情况下的法向量。

这就涉及了折算的问题，最直观的方式就是根据片元处的法向量、切向量将纹理图中取出的扰动后法向量折算到实际坐标系中进行光照计算。但基于这种思路的计算非常复杂，故为了简化问题，本节案例采用的是逆向计算的方式，实施策略为将光源位置、摄像机位置变换到标准法向量所属的坐标系中再直接基于从纹理图中取出的扰动后法向量进行光照计算，具体步骤如下。

- 首先根据片元的法向量、切向量计算出副法向量。副法向量的计算很简单，将法向量、切向量直接进行叉积即可得到。
- 然后将切向量、副法向量、法向量组装成一个3x3的变换矩阵，此变换矩阵可以用来将光源位置、摄像机位置变换到标准法向量所属的坐标系中。
- 将光源位置乘以上述3x3的变换矩阵，得到用于光照计算的光源位置。

- 将摄像机位置乘以上述3x3的变换矩阵，得到用于光照计算的摄像机位置。

> **提示**
>
> 有了上述计算模型后，就可以开发出实现此计算模型的渲染程序了，具体内容会在后面的小节进行介绍。

15.6.2 法向量纹理图的生成

从上一小节凹凸映射计算模型的介绍中可以看出，实施此计算模型需要有携带了扰动后法向量信息的纹理图。具体来说就是纹理图中的每个像素都代表了一个扰动后的法向量，RGB通道的值分别为扰动后法向量的X、Y、Z分量。

显然，这种纹理图是没有办法用绘图工具（如PhotoShop、画图等）来直接绘制的。笔者采用的方式是用绘图工具生成高度域灰度图，然后再用自己开发的工具将高度域灰度图转换为法向量纹理图。图15-15给出了一系列的高度域灰度图。



图15-15　高度域灰度图

> **提示**
>
> 图15-15中给出了4幅高度域灰度图，其中最左侧的是本节案例所采用的，读者掌握了开发技术之后也可以采用其他高度域灰度图。

高度域灰度图中黑色的部分代表高度低的区域，而白色的部分代表凸起的区域。也就是颜色越浅，凸起越高。有了高度域灰度图后就可以将其转换为法向量纹理图以备使用了，具体转换方法如下所列。

- 当需要计算某个像素对应的扰动后法向量时，首先需要计算出两个差分向量。第一个差分向量为[1，0，$H_{上}-H_{本}$]，第二个差分向量为[0，1，$H_{右}-H_{本}$]。$H_{上}$为此像素正上方的像素灰度值所代表的高度，$H_{右}$为此像素正右侧的像素灰度值所代表的高度，$H_{本}$为此像素灰度值所代表的高度。

提示

$H_{上}$、$H_{右}$、$H_{本}$并不能从高度域灰度图中直接取出，当取出一个像素的灰度值后需要除以255以折算出0~1内的高度。

- 计算出两个差分向量后将两个差分向量进行叉积即可计算出此像素处扰动后法向量的值，然后再将扰动后法向量的X、Y、Z分量各自折算到0～255的整数范围内，并作为像素的RGB值存入法向量纹理图中的对应位置即可。

了解了法向量纹理图的生成算法后，就可以基于此开发出将高度域灰度图转换为法向量纹理图的工具了。在介绍转换工具的开发之前可以首先了解一下转换后法向量纹理图的内容，具体情况如图15-16所示。

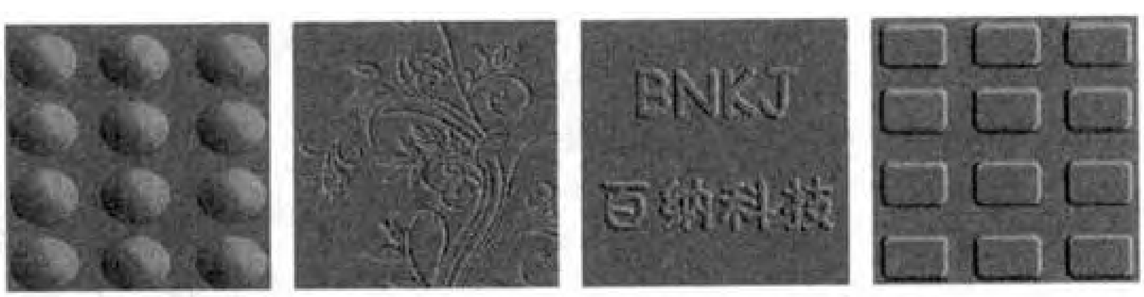


图15-16　生成的法向量纹理图

说明

图15-16中给出了图15-15中4幅高度域灰度图对应的法向量纹理图。要注意的是由于书中的插图是灰度印刷的因此很难看出效果，若读者运行光盘中附带的法向量纹理图自动生成工具就会发现生成的法向量纹理图其主体颜色是蓝色，在高度变化大的区域会有其他颜色。

了解了法向量纹理图的具体内容后，就可以进行转换工具的开发了。由于本转换工具是采用Java进行开发，工作在PC上，大部分搭建界面和人机交互的代码与本书内容无关，故略去不讲，需要的读者可以参考随书关盘中的源代码。这里仅简要介绍一下实现此生成算法的核心代码，具体内容如下所列。

（1）首先介绍对向量进行各种运算的工具类VectorUtil，其代码如下。

代码位置：见随书光盘中源代码/第15章/高度域灰度图转换为法向量纹理图的工具目录下的VectorUtil.java。

```
package com.bn;
public class VectorUtil{
    public static float[] getCrossProduct                    //求两个向量叉积的方法
    (float x1,float y1,float z1,float x2,float y2,float z2){
        //求出两个矢量的叉积矢量在X、Y、Z轴上的分量A、B、C
        float A=y1*z2-y2*z1;  float B=z1*x2-z2*x1;  float C=x1*y2-x2*y1;
        return new float[]{A,B,C};                           //返回结果向量
    }
    public static float[] vectorNormal(float[] vector){          //向量规格化的方法
            float module=(float)Math.sqrt(vector[0]*vector[0]    //求向量的模
                        +vector[1]*vector[1]+vector[2]*vector[2]);
            return new float[]{vector[0]/module,vector[1]/module,vector[2]/module};
      }
      public static float mould(float[] vec){                    //求向量模的方法
            return (float)Math.sqrt(vec[0]*vec[0]+vec[1]*vec[1]+vec[2]*vec[2]);
      }
}
```

说明

上述VectorUtil类中主要包含了计算向量的模、向量规格化及计算两个向量叉积的方法。如果读者对于向量相关的数学不是很熟悉，请查阅其他相关的技术资料。

（2）接着介绍的是实现将高度域灰度图转换为法向量纹理图核心算法的方法process，其代码如下。

代码位置：见随书光盘中源代码/第15章/高度域灰度图转换为法向量纹理图的工具目录下的NormalMapUtil.java。

```java
public Image process(){
    int width=ii.getImage().getWidth(null);//获取待处理图像的宽度与高度
    int height=ii.getImage().getHeight(null);
    //创建两个 BufferedImage 对象分别用来放置待处理图像与处理后的图像
    BufferedImage sourceBuf=new BufferedImage(width,height,BufferedImage.TYPE_INT_ARGB);
    BufferedImage targetBuf=new BufferedImage(width,height,BufferedImage.TYPE_INT_RGB);
    Graphics graph=sourceBuf.getGraphics();//将待处理图像绘制加载到源 BufferedImage 对象中
    graph.drawImage(ii.getImage(),0,0,Color.white,null);
    for(int i=0;i<height;i++){//对待处理图中像素的行循环
        for(int j=0;j<width;j++){//对待处理图中像素的列循环
            int color=sourceBuf.getRGB(j,i);//获取指定位置处的像素
            //拆分出 RGB 3 个色彩通道的值
            int r=(color >> 16) & 0xff;int g=(color >> 8) & 0xff;int b=(color) & 0xff;
            float c=(r+g+b)/3.0f/255.0f;   //求出折算后此像素的高度
            if(i==0||j==width-1){          //若为最左侧一列或最上面一行的像素不用计算
                targetBuf.setRGB(j,i,0xFF8080FF);
                continue;
            }
            //取出正上方像素的值并折算成高度
            int colorUp=sourceBuf.getRGB(j,i-1);
            int rUp=(colorUp >> 16) & 0xff; int gUp=(colorUp >> 8) & 0xff; int bUp=(colorUp) & 0xff;
            float cUp=(rUp+gUp+bUp)/3.0f/255.0f;
            //取出正右侧像素的值并折算成高度
            int colorRight=sourceBuf.getRGB(j+1,i);
            int rRight=(colorRight >> 16) & 0xff; int gRight=(colorRight >> 8) & 0xff;
            int bRight=(colorRight) & 0xff;
            float cRight=(rRight+gRight+bRight)/3.0f/255.0f;
            //计算出两个差分向量
            float[] vec1={1,0,cUp-c}; float[] vec2={0,1,cRight-c};
            float[] vResult=VectorUtil.getCrossProduct(   //将差分向量叉积得到结果向量
                vec1[0],vec1[1],vec1[2]*4,vec2[0],vec2[1],vec2[2]*4);
            vResult=VectorUtil.vectorNormal(vResult);
            //将结果向量各分量值折算到 0～255 内
            int cResultRed=(int)(vResult[0]*128)+128;
            int cResultGreen=(int)(vResult[1]*128)+128;
            int cResultBlue=(int)(vResult[2]*128)+128;
            cResultRed=(cResultRed>255)?255:cResultRed;
            cResultGreen=(cResultGreen>255)?255:cResultGreen;
            cResultBlue=(cResultBlue>255)?255:cResultBlue;
            //将结果向量送入像素
            int cResult=0xFF000000;
            cResult+=cResultRed<<16; cResult+=cResultGreen<<8; cResult+=cResultBlue;
            targetBuf.setRGB(j,i,cResult);
        }
    }
    return targetBuf;                              //返回结果
}
```

> **提示**
>
> 上述代码实现了前面介绍的将高度域灰度图转换成法向量纹理图的算法，具体情况在注释中有详细的说明。

了解了转换工具的核心代码后，下面简单介绍一下此工具的使用，运行工具中的主类NormalMapUtil，将出现如图15-17所示的界面，要求用户选择一幅高度域灰度图。选择完毕单击打开按钮后，将出现如图15-18所示的结果界面，同时程序也将结果法向量纹理图存入磁盘下指定的resultnt.jpg文件中。



图15-17　文件选择界面

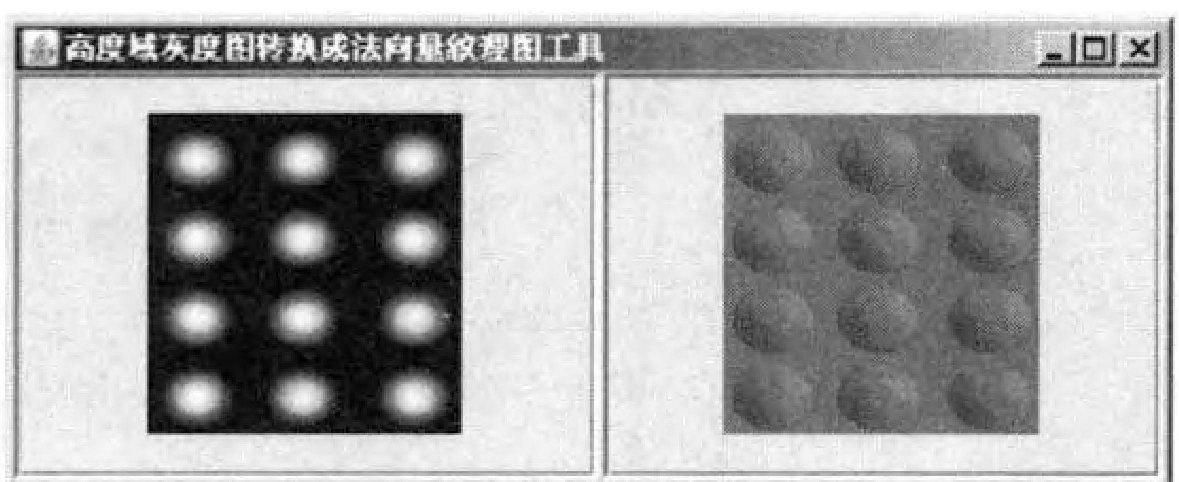


图15-18　转换结果界面

15.6.3 案例的开发

介绍完了凹凸映射的基本原理以及法向量纹理图的生成后，就可以正式进行本案例的开发了。本案例主要是基于前面介绍模型加载的章节中最后一个案例（加载的模型中带有纹理坐标的）升级而成的，因此，这里仅介绍代码改动较大且有代表性的部分，具体内容如下所列。

（1）前面已经介绍过，凹凸映射计算中进行变换时需要用到切向量，其是由Java程序传入渲染管线的，因此在加载模型的工具类LoadUtil中，首先需要增加计算切向量的工具方法fromNormalToTangent，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_6/com/bn/Sample15_6目录下的LoadUtil.java。

```java
public static float[] fromNormalToTangent(float[] normal,float[] position){
    //取出求切面点法式平面方程需要的参数
    float A=normal[0];float B=normal[1];float C=normal[2];
    float x0=position[0];float y0=position[1];float z0=position[2];
    float[] resultY=null;                    //声明用于存放第一个切向量各分量值的数组
    //求出切平面上离position位置X、Z分量各差1个单位的点的坐标
    float x1=x0+1; float z1=z0+1; float y1=(C*(z0-z1)+A*(x0-x1))/B+y0;
    resultY=new float[]{x1-x0,y1-y0,z1-z0};//求出第一个切向量
    float[] resultZ=null;                    //声明用于存放第二个切向量各分量值的数组
        //求出切平面上离position位置X、Y分量各差1个单位的点的坐标
        x1=x0+1;y1=y0+1; z1=(A*(x0-x1)+B*(y0-y1))/C+z0;
        resultZ=new float[]{x1-x0,y1-y0,z1-z0};//求出第二个切向量
        float[] resultX=null;                //声明用于存放第三个切向量各分量值的数组
        //求出切平面上离position位置Y、Z分量各差1个单位的点的坐标
        y1=y0+1; z1=z0+1; x1=(B*(y0-y1)+C*(z0-z1))/A+x0;
        resultX=new float[]{x1-x0,y1-y0,z1-z0}; //求出第三个切向量
        if(resultX[0]<50){return resultX;}      //若第一个切向量在指定范围内则取第一个
        else if(resultY[1]<50){return resultY;}//若第二个切向量在指定范围内则取第二个
        else{return resultZ;}                  //否则取第三个
}
```

说明

切向量与法向量不同，平滑物体表面上的某点仅有一个法向量，但有无数个切向量，这些切向量都位于切平面中。本方法首先通过传入的点坐标位置及法向量的3个分量得到点法式平面方程需要的各个参数，然后根据点法式方程求得3个不同方向的切向量，

并取3个向量中值良好的作为选中的切向量返回。点法式是空间解析几何中的一种平面表示方法，具体的情况请读者参阅其他相关资料。

（2）完成了fromNormalToTangent工具方法的开发后，就可在LoadUtil类中执行加载任务的loadFromFile方法中增加计算切向量的相关代码，具体内容如下。

代码位置：见随书光盘中源代码/第15章/Sample15_6/com/bn/Sample15_6目录下的LoadUtil.java。

```
public static LoadedObjectVertexNormalTexture loadFromFile
(String fname, Resources r,MySurfaceView mv) {
  ……//此处省略了很多与原来案例相同的代码，需要的读者请参考随书光盘
  float[] tnXYZ=new float[nXYZ.length];            //创建存放顶点切向量数据的数组
  int kc=nXYZ.length/3;                            //顶点数量
  for(int i=0;i<kc;i++) {                          //对每个顶点循环，计算切向量
   float[] normal=new float[]{nXYZ[i*3],nXYZ[i*3+1],nXYZ[i*3+2]};    //获取顶点法向量
   float[] position=new float[]{vXYZ[i*3],vXYZ[i*3+1],vXYZ[i*3+2]}; //获取顶点坐标
   float[] tangent=fromNormalToTangent(normal,position);//通过顶点坐标与法向量计算切向量
      //将切向量存入存放顶点切向量数据的数组
      tnXYZ[i*3]=tangent[0];tnXYZ[i*3+1]=tangent[1]; tnXYZ[i*3+2]=tangent[2];
     }
     ……//此处省略了一些与原来案例相同的代码，需要的读者请参考随书光盘
     lo=new LoadedObjectVertexNormalTexture(mv,vXYZ,nXYZ,tST,tnXYZ);//创建加载的物体对象
     ……//此处省略了一些与原来案例相同的代码，需要的读者请参考随书光盘
      return lo;                                         //返回加载后的对象
  }
```

说明

上述方法中主要增加了对每个顶点进行循环，通过顶点的位置坐标及法向量数据计算出切向量数据的代码。最后通过顶点位置数据、顶点法向量数据、顶点切向量数据创建了加载的物体对象，并将加载后物体对象的引用返回。

（3）从上述代码的改动中可以看出，LoadedObjectVertexNormalTexture类的构造器增加了用于接收顶点切

向量数据的参数，改动后LoadedObjectVertexNormalTexture类的代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_6/com/bn/Sample15_6目录下的LoadedObjectVertexNormalTexture.java。

```
package com.bn.gles20.ex42;
......//省略了一些 import 语句
public class LoadedObjectVertexNormalTexture{
......//省略了很多与前面案例相同的成员变量声明，需要的读者请参考随书光盘
 int maTangentHandle;                  //顶点切向量属性引用
 FloatBuffer   mTangentBuffer;         //顶点切向量数据缓冲
 int uTexHandle;                       //外观纹理一致变量引用
 int uNormalTexHandle;                 //法向量纹理一致变量引用
   public LoadedObjectVertexNormalTexture(
        MySurfaceView mv,float[] vertices,float[] normals,float[] texCoors,float[]
        tangent){
        initVertexData(vertices,normals,texCoors,tangent);   //初始化顶点坐标与着色数据
        initShader(mv);                                      //初始化 shader
      }
      public void initVertexData(float[] vertices,      //初始化顶点坐标与着色数据的方法
      float[] normals,float texCoors[],float[] tangent) {
          ......//省略了很多与前面案例相同的成员变量声明，需要的读者请参考随书光盘
          //将顶点切向量送入缓冲
          ByteBuffer tnbb = ByteBuffer.allocateDirect(tangent.length*4);
          tnbb.order(ByteOrder.nativeOrder());           //设置字节顺序
          mTangentBuffer = tnbb.asFloatBuffer();         //转换为 Float 型缓冲
```

```
            mTangentBuffer.put(tangent);                //向缓冲区中放入顶点切向量数据
            mTangentBuffer.position(0);                 //设置缓冲区起始位置
        }
        public void intShader(MySurfaceView mv) {      //初始化着色器的方法
            ……//省略了很多与前面案例相同的成员变量声明，需要的读者请参考随书光盘
            //获取顶点切向量属性引用
            maTangentHandle= GLES20.glGetAttribLocation(mProgram, "tNormal");
            //获取外观、法线两个纹理一致变量引用
            uTexHandle=GLES20.glGetUniformLocation(mProgram, "sTextureWg");
            uNormalTexHandle=GLES20.glGetUniformLocation(mProgram, "sTextureNormal");
        }
        public void drawSelf(int texId,int texIdNormal) {      //绘制物体的方法
             ……//省略了很多与前面案例相同的成员变量声明，需要的读者请参考随书光盘
             GLES20.glVertexAttribPointer (maTangentHandle,3, //将顶点切向量数据送入渲染管线
             GLES20.GL_FLOAT,false,3*4, mTangentBuffer);
             GLES20.glEnableVertexAttribArray(maPositionHandle);   //启用顶点位置数据
             GLES20.glEnableVertexAttribArray(maNormalHandle) ;    //启用顶点法向量数据
             GLES20.glEnableVertexAttribArray(maTangentHandle);    //启用顶点切向量数据
             GLES20.glEnableVertexAttribArray(maTexCoorHandle);    //启用顶点纹理坐标数据
             GLES20.glActiveTexture(GLES20.GL_TEXTURE0);           //启用 0 号纹理
             GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, texId);    //绑定外观纹理
             GLES20.glActiveTexture(GLES20.GL_TEXTURE1);           //启用 1 号纹理
             GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, texIdNormal); //绑定法向量纹理
             GLES20.glUniform1i(uTexHandle, 0);               //通过引用指定外观纹理
             GLES20.glUniform1i(uNormalTexHandle, 1);         //通过引用指定法向量纹理
             GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount); //绘制加载的物体
        }
}
```

说明

从上述代码中可以看出，主要是增加了将顶点切向量及法向量纹理送入渲染管线供着色器使用的代码，其他部分与前面章节中加载模型的案例基本相同。

（4）完成了LoadedObjectVertexNormalTexture类的修改后，Java代码的修改就基本完成了。下面需要修改的就是着色器的代码了，首先是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_6/assets目录下的vertex_ut.sh。

```
uniform mat4 uMVPMatrix;          //总变换矩阵
attribute vec3 aPosition;         //顶点位置
attribute vec2 aTexCoor;          //顶点纹理坐标
attribute vec3 aNormal;           //法向量
attribute vec3 tNormal;           //切向量
varying vec2 vTextureCoord;       //用于传递给片元着色器的纹理坐标
varying vec3 fNormal;             //用于传递给片元着色器的法向量
varying vec3 ftNormal;            //用于传递给片元着色器的切向量
varying vec3 vPosition;           //用于传递给片元着色器的顶点位置
  void main() {
    gl_Position = uMVPMatrix * vec4(aPosition,1);     //根据总变换矩阵计算此次绘制此顶
                                                      //点的位置
     vTextureCoord=aTexCoor;                  //将顶点的纹理坐标传给片元着色器
     fNormal=aNormal;                         //将顶点的法向量传给片元着色器
     ftNormal=tNormal;                        //将顶点的切向量传给片元着色器
     vPosition=aPosition;                     //将顶点的位置传给片元着色器
  }
```

说明

上述顶点着色器中首先增加了从管线接收切向量属性变量值的代码，然后增加了将切向量值从顶点着色器传递给片元着色器的代码。

（5）完成了顶点着色器的修改后，就可以修改片元着色器的代码了，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_6/assets目录下的frag_ut.sh。

```
precision mediump float;                          //给出默认浮点精度
varying vec2 vTextureCoord;                       //接收从顶点着色器传递过来的纹理坐标
varying vec3 fNormal;                             //接收从顶点着色器传递过来的法向量
varying vec3 ftNormal;                            //接收从顶点着色器传递过来的切向量
varying vec3 vPosition;                           //接收从顶点着色器传递过来的顶点位置
uniform sampler2D sTextureWg;                     //纹理内容数据（外观）
uniform sampler2D sTextureNormal;                 //纹理内容数据（法线）
uniform mat4 uMMatrix;                            //变换矩阵
uniform vec3 uCamera;                             //摄像机位置
   uniform vec3 uLightLocationSun;                //光源位置
   void pointLight(                               //定位光光照计算的方法
    in vec3 normal,                               //扰动后法向量
    out vec4 ambient,                             //最终环境光强度
    out vec4 diffuse,                             //最终散射光强度
    out vec4 specular,                            //最终镜面光强度
    in vec3 vp,                                   //变换到标准法向量所属坐标系的表面点到光源位置的向量
    in vec3 eye,                                  //变换到标准法向量所属坐标系的视线向量
    in vec4 lightAmbient,                         //环境光强度
    in vec4 lightDiffuse,                         //散射光强度
    in vec4 lightSpecular                         //镜面光强度
   ){
    ambient=lightAmbient;                                  //直接得出环境光的最终强度
    vec3 halfVector=normalize(vp+eye);                     //求视线与光线的半向量
    float shininess=50.0;                                  //粗糙度，越小越光滑
    float nDotViewPosition=max(0.0,dot(normal,vp)); //求法向量与 vp 的点积与 0 的最大值
    diffuse=lightDiffuse*nDotViewPosition;                 //计算散射光的最终强度
    float nDotViewHalfVector=dot(normal,halfVector);  //法向量与半向量的点积
    float powerFactor=max(0.0,pow(nDotViewHalfVector,shininess)); //镜面反射光强度因子
    specular=lightSpecular*powerFactor;                    //计算镜面光的最终强度
   }
   void main(){
     vec4 ambient,diffuse,specular;    //用来接收光 3 个通道最终强度的变量
     vec4 normalColor = texture2D(sTextureNormal, vTextureCoord); //从法线纹理图中读出值
```

```
34      //将值恢复到-1－+1
35      vec3 cNormal=vec3(2.0*(normalColor.r-0.5),2.0*(normalColor.g-0.5),2.0*
        (normalColor.b-0.5));
36      cNormal=normalize(cNormal);   //将扰动结果向量规格化
37      //计算变换后的法向量
38      vec3 normalTarget=vPosition+fNormal;
39      vec3 newNormal=(uMMatrix*vec4(normalTarget,1)).xyz-(uMMatrix*vec4(vPosition,
        1)).xyz;
40      newNormal=normalize(newNormal);
41      //计算变换后的切向量
42      vec3 tangentTarget=vPosition+ftNormal;
43      vec3 newTangent=(uMMatrix*vec4(tangentTarget,1)).xyz-(uMMatrix*vec4(vPosition,
        1)).xyz;
44      newTangent=normalize(newTangent);
45      vec3 binormal=normalize(cross(newTangent,newNormal)); //计算副法向量
46      //用切向量、副法向量、法向量搭建变换矩阵，此矩阵用于将向量从实际坐标系
47      //变换到标准法向量所属的坐标系
48      mat3 rotation=mat3(newTangent,binormal,newNormal);
49      vec3 newPosition=(uMMatrix*vec4(vPosition,1)).xyz;     //变换后的片元位置
50      vec3 vp= normalize(uLightLocationSun-newPosition);//求表面点到光源位置的向量 vp 并规格化
51      vp=normalize(rotation*vp);                        //变换并规格化 vp 向量
52      //求出从表面点到摄像机的视线向量然后进行变换并规格化
53      vec3 eye= normalize(rotation*normalize(uCamera-newPosition));
54      pointLight(cNormal,ambient,diffuse,specular,vp,eye,
55        vec4(0.05,0.05,0.05,1.0),vec4(1.0,1.0,1.0,1.0),vec4(0.3,0.3,0.3,1.0));
56      vec4 finalColor=texture2D(sTextureWg, vTextureCoord);//根据纹理坐标采样出片元颜色值
57      //综合 3 个通道光的最终强度及片元的颜色计算出最终片元的颜色并传递给渲染管线
58      gl_FragColor = finalColor*ambient+finalColor*specular+finalColor*diffuse;
59   }
```

- 第11-30行的pointLight方法有了一些变化，接收的参数增加了视线向量与表面点到光源位置的向量。随着参数的变化，计算也简单了一些，方法中不再需要计算视线向量与表面点到光源位置的向量了，总体计算思路不变。
- 第31-59行的main方法除了完成原来的任务外，还增加了从法向量纹理图中读出扰动后法向量，并将视线向量、表面点到光源位置的向量变换到标准法向量所属坐标系的代码。

说明

总的来说，上述片元着色器代码实现了15.6.1小节中介绍的凹凸映射算法。另外，上述着色器具有一定的通用性，读者需要开发出凹凸映射效果的场景时可以直接使用。

到此为止如何通过凹凸映射技术渲染表面具有细节结构的物体就介绍完了，最后需要提醒读者注意的是：只有在应用光照的场合凹凸

映射才能起作用，因为其本质上并没有实现表面的细节几何结构。

15.7　平面阴影

前面第6章介绍光照时已经提到过，由于采用的光照模型是基于每个顶点或每个片元独立计算 的，计算时仅仅考虑了光源位置、法向量、摄像机位置与被照射点位置，并没有考虑其他物体的存在（其实最准确的说是没有考虑其他顶点或片元的存在），所以是不会产生阴影效果的。

但在很多场景中没有了阴影后真实感会差很多，很多著名的3D游戏中也都有真实的阴影效果。从本节开始，将介绍几种基于OpenGL ES 2.0的阴影生成技术，供读者在开发中根据情况选用。

15.7.1　案例效果与基本原理

本节介绍的是阴影生成技术中最简单的一种——平面阴影，其易于理解和开发，性能也尚可。下面首先了解一下本节案例Sample15_7的运行效果，如图15-19所示。



图15-19　平面阴影案例运行效果

说明

图15-19中最左侧是光源从右上侧照射场景的情况，中间是光源从场景前上侧照射的情况，右侧是光源从场景左上侧照射的情况。另外本案例运行时，当手指在屏幕上水平滑动时摄像机会绕场景转动，当手指在屏幕上垂直滑动时摄像机会随手指的移动而升降。

本节案例中采用的阴影生成算法非常简单，其基本思想为，根据照射光线的方向将物体投影到需要显示阴影的平面上，具体情况如图15-20所示。

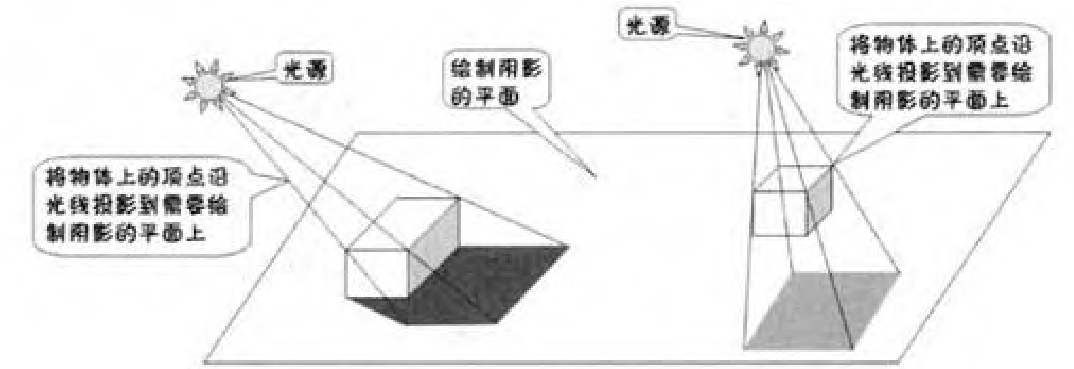


图15-20　平面阴影的基本原理

从图15-20中可以看出，平面阴影技术中最关键的就是如何实现把物体上的顶点沿光线投影到绘制阴影的平面上。读者可能很自然地会想到利用空间解析几何的知识来进行计算，但那样太复杂了，实现起来会很困难，运行速度也会很慢。

真正实现的时候还是可以继续依赖向量的计算，具体的计算公式如下所列。

$$V' =S+(V-S)(n\cdot(A-S))/(n\cdot(V-S))$$

上述公式中每一部分的含义如下。

- V’代表沿光线投影到绘制阴影的平面上顶点的坐标。
- S为光源的位置。
- V为投影前顶点的实际位置。
- n为法向量，其属于绘制阴影的平面。
- A为任意一个点的坐标，其位于绘制阴影的平面上。
- •代表向量的点积计算。

> **提示**
>
> 实际计算时上述这些量都是在着色器中用vec3类型的变量来实现，具体情况下一小节会进行详细介绍。

15.7.2　开发步骤

了解了案例的运行效果及基本原理后，就可以进行案例的开发了，具体步骤如下。

（1）首先用3DMax生成5个基本物体（平面、圆环、茶壶、立方体、圆球），并导出成obj文件放入项目的assets目录待用。

（2）开发出搭建场景的基本代码，包括：加载物体、摆放物体、计算光照等。这些代码与前面章节的许多案例基本套路完全一致，因此这里不再赘述。

提示

需要注意的是，为了达到更好的光照效果，立方体和平面使用的加载方法采用的是面法向量，而圆环、茶壶和圆球使用的加载方法采用的是点平均法向量，因此用于完成加载的工具类LoadUtil中就有两个不同的加载方法，loadFromFileVertexOnlyAverage与loadFromFileVertexOnlyFace。其中第一个采用点平均法向量，第二个采用面法向量，代码与第9章案例中的相同。

（3）采用平面阴影技术实施阴影时，需要阴影的物体应该绘制两次，第一次绘制阴影，第二次绘制物体本身。因此MySurfaceView中绘制场景的代码也会有所不同，具体内容如下。

代码位置：见随书光盘中源代码/第15章/Sample15_7com/bn/Sample15_7目录下的MySurfaceView.java。

```
public void onDrawFrame(GL10 gl) {
     ……//此处省略了设置摄像机、光源与清除缓冲的代码，与前面很多案例中的相同
   pm.drawSelf(0); //绘制平面
   drawObject(1);  //绘制平面上各个物体的阴影
   drawObject(0);  //绘制物体本身
}
public void drawObject( //绘制平面上物体的方法
         int situ) {  //根据参数的不同决定绘制阴影还是物体本身
      MatrixState.pushMatrix();                       //保护现场
      MatrixState.scale(1.5f, 1.5f, 1.5f);            //进行缩放变换
      MatrixState.translate(-10f, 0f, 0);             //进行平移变换
      cft.drawSelf(situ);                             //绘制长方体
      MatrixState.popMatrix();                        //恢复现场
```

```
            ......//此处省略球体、圆环和茶壶的绘制代码，方法与绘制长方体的相同
            ......//需要的读者请查看随书光盘中的源代码
    }}
```

说明

上述代码中用于绘制场景中物体的方法drawObject需要接收一个整型的参数，当参数值为1时绘制物体在指定平面上的阴影，参数值为0时绘制物体本身。

（4）由于加载物体时根据表面情况的不同采用了不同的法向量策略，因此表示加载后物体的类就有两个，LoadedObjectVertexNormalAverage与LoadedObjectVertexNormalFace。其中第一个表示采用点平均法向量的物体，第二个表示采用面法向量的物体。两个类的代码都与第9章对应案例中的基本一致，主要的区别就是增加了将阴影绘制标志以及投影、摄像机组合矩阵传入渲染管线的相关代码，下面给出了LoadedObjectVertexNormalFace类的代码。

代码位置：见随书光盘中源代码/第15章/Sample15_7com/bn/Sample15_7目录下的LoadedObjectVertexNormalFace.java。

```java
package com.bn.Sample9_3;
……//此处省略了部分类的引入代码，读者可自行查看随书光盘中的源代码
public class LoadedObjectVertexNormalFace {//采用面法向量的加载物体
  ……//此处省略了部分成员变量的声明，读者可自行查看随书光盘中的源代码
  int muIsShadow;                              //阴影绘制标志对应的一致变量引用
  int muProjCameraMatrixHandle;                //投影、摄像机组合矩阵引用
  public LoadedObjectVertexNormalFace(MySurfaceView mv,     //构造器
    float[] vertices,float[] normals) {/*代码省略*/ }
  public void initVertexData(float[] vertices,float[] normals){/*代码省略*/ }
      public void initShader(MySurfaceView mv){//初始化shader
          ……//此处省略了一些代码，与第 9 章相关案例中的相同
          //获取着色器中阴影绘制标志对应的一致变量引用
          muIsShadow=GLES20.glGetUniformLocation(mProgram, "isShadow");
          muProjCameraMatrixHandle=           //获取程序中投影、摄像机组合矩阵的引用
             GLES20.glGetUniformLocation(mProgram, "uMProjCameraMatrix");
      }
      public void drawSelf(int isShadow)  {    //绘制物体的方法
          ……//此处省略了一些代码，与第 9 章相关案例中的相同
          GLES20.glUniform1i(muIsShadow, isShadow); //将阴影绘制标志传入 shader 程序
          GLES20.glUniformMatrix4fv(       //将投影、摄像机组合矩阵传入 shader 程序
          muProjCameraMatrixHandle, 1, false, MatrixState.getViewProjMatrix(),0);
          ……//此处省略了一些代码，与第 9 章相关案例中的相同
      }
  }
```

提示

LoadedObjectVertexNormalAverage类的变化与LoadedObjectVertexNormalFace类几乎完全一样，因此这里不再赘述，需要的读者请查看随书光盘中的源代码。

（5）完成了Java代码的开发后就可以开发着色器了，首先是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_7/assets目录下的vertex_shadow.sh。

```
1 uniform int isShadow;                    //阴影绘制标志
2 uniform mat4 uMVPMatrix;                 //总变换矩阵
3 uniform mat4 uMMatrix;                   //变换矩阵
4 uniform mat4 uMProjCameraMatrix;         //投影、摄像机组合矩阵
5 uniform vec3 uLightLocation;             //光源位置
6 uniform vec3 uCamera;                    //摄像机位置
7 attribute vec3 aPosition;                //顶点位置
8 attribute vec3 aNormal;                  //顶点法向量
9 varying vec4 ambient;                    //用于传递给片元着色器的环境光最终强度
10    varying vec4 diffuse;                //用于传递给片元着色器的散射光最终强度
11    varying vec4 specular;               //用于传递给片元着色器的镜面光最终强度
12    ……//此处省略了定位光光照计算的方法pointLight，代码与前面很多案例中的相同
13    void main(){
14     if(isShadow==1) {                   //标志位为1则绘制阴影
15      vec3 A=vec3(0.0,0.1,0.0);          //绘制阴影的平面上任意一点的坐标
16      vec3 n=vec3(0.0,1.0,0.0);          //绘制阴影平面的法向量
17      vec3 S=uLightLocation;             //光源位置
18      vec3 V=(uMMatrix*vec4(aPosition,1)).xyz;  //经过平移和旋转变换后的点的坐标
19     vec3 VL=S+(V-S)*(dot(n,(A-S))/dot(n,(V-S)));//顶点沿光线投影到需要绘制阴影的平面上点的坐标
20         //根据投影、摄像机组合矩阵计算此次绘制此顶点位置
21         gl_Position = uMProjCameraMatrix*vec4(VL,1);
22      } else {
23         gl_Position = uMVPMatrix * vec4(aPosition,1);//根据总变换矩阵计算此次绘制此顶点位置
24      }
25      pointLight(normalize(aNormal),ambient,diffuse,     //计算光照
26     specular,uLightLocation,vec4(0.4,0.4,0.4,1.0),vec4(0.7,0.7,0.7,1.0),vec4(0.3,0.3,0.3,
1.0));
27    }
```

- 第14-21行为根据上一小节介绍的平面阴影计算模型计算投影点位置的代码，其中绘制阴影平面的法向量和任意一点坐标是固化在着色器中的，未来有需要也可以从Java程序中传入。
- 第21行将计算出的投影点坐标用投影、摄像机组合矩阵进行变换得到最终此顶点的绘制位置。有的读者可能会有这样的疑问：“为什么不直接用总变换矩阵计算呢？”这是因为总变换矩阵不但携带了投影、摄像机观察的变换信息，还携带了平移、旋转、缩放变换的信息，所以只应该用于对原始顶点坐标进行变换。而投影点的位置已经确定，不应该进行平移、旋转、缩放等基本变换了，其仅仅需要用投影及摄像机观察矩阵进行变换即可得出最终绘制位置。
- 第25-26行调用pointLight方法进行了光照计算。

提示

从上述代码中可以看出，由于着色语言原生支持向量的计算，所以采用其实现上一小节的投影计算公式非常容易。

（6）完成了顶点着色器后就应该开发片元着色器了，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_7/assets目录下的frag_shadow.sh。

```
precision mediump float;                    //给出默认的浮点精度
uniform highp int isShadow;                 //阴影绘制标志
varying vec4 ambient;                       //从顶点着色器传递过来的环境光最终强度
varying vec4 diffuse;                       //从顶点着色器传递过来的散射光最终强度
varying vec4 specular;                      //从顶点着色器传递过来的镜面光最终强度
void main() {
    if(isShadow==0){                                          //绘制物体本身
        vec4 finalColor=vec4(1.0,1.0,1.0,0.0);                //物体本身的颜色
        //综合3个通道光的最终强度及片元的颜色计算出最终片元的颜色并传递给管线
            gl_FragColor = finalColor*ambient+finalColor*specular+finalColor*diffuse;
     }else{                                                   //绘制阴影
           gl_FragColor = vec4(0.2,0.2,0.2,0.0);;             //片元最终颜色为阴影的颜色
     }
}
```

说明

从上述代码中可以看出，此片元着色器根据阴影绘制标志的值决定片元采用哪种颜色着色。若阴影标志值为0，则根据物体本身的颜色与3个光照通道强度加权计算出片元最终颜色；若阴影标志值为1，则直接采用阴影颜色作为片元的最终颜色。

到这里为止平面阴影技术就介绍完了，在以后的具体项目开发中，读者若能做到灵活运用，将可以开发出更加逼真的场景。

15.8　阴影映射

上一节介绍了平面阴影实现技术，从其案例的场景抓图中可以看出效果还是比较真实的。但细心的读者应该能够发现一个明显的瑕疵，那就是基于此技术的阴影仅能出现在目标平面上，无论光源处于什么位置，目标平面上的物体永远不会处于阴影中。

上述显然不符合现实世界的情况，因此本节将介绍另外一种阴影实现技术——阴影映射。采用阴影映射技术可以方便地实现物体相互遮挡时的阴影，开发出来的场景效果会更加真实。

15.8.1　案例效果与基本原理

正式介绍阴影映射技术的原理与具体开发步骤之前，请读者首先了解一下本节案例的运行效果，如图15-21所示。

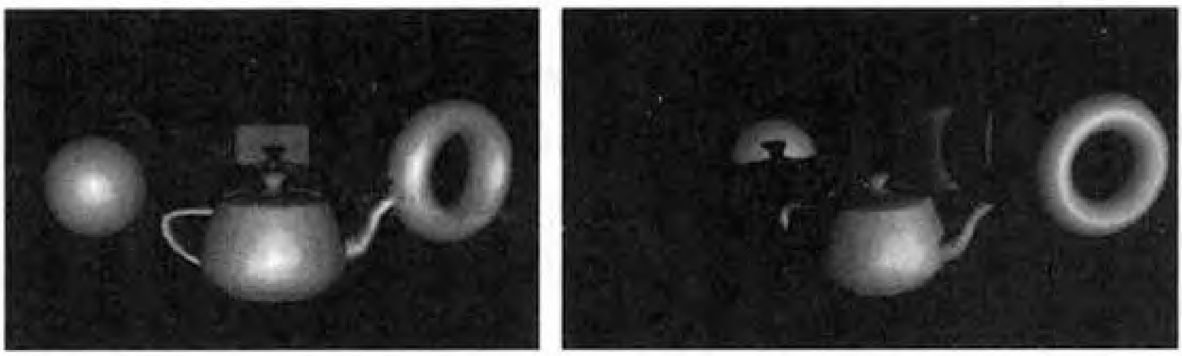


图15-21　阴影映射案例运行效果

> **说明**
>
> 本案例中光源是围绕场景旋转的，因此阴影也会随光源的运动而移动。图15-21中左侧为光源从摄像机后面照射场景的情况，茶壶形状的阴影映射到了其前面的立方体上；右侧为光源从摄像机右后侧照射的情况，茶壶形状的阴影映射到了其左前方的圆球上。与上一节的案例比较即可看出，此技术实现的阴影其真实感更好，更接近现实世界的情况。

本节案例采用的阴影生成算法比上一节的要复杂一些，需要经过两次绘制，还要采用投影贴图技术才能实现，其基本原理如图15-22所示。

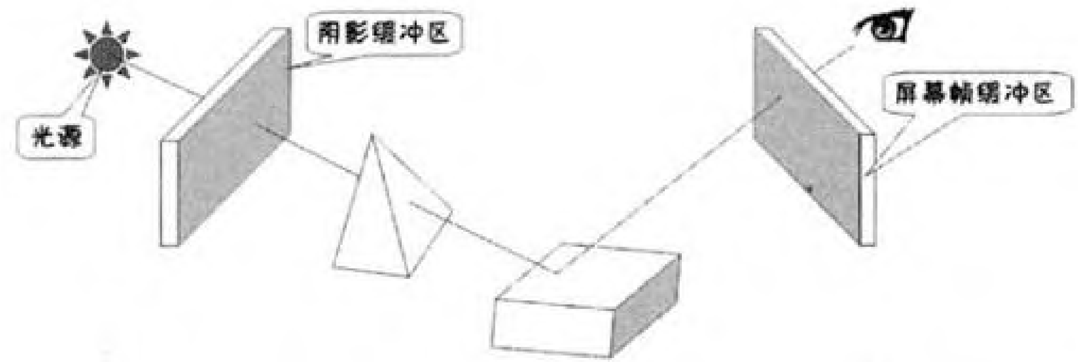


图15-22　阴影映射的基本原理

从图15-22中可以看出，阴影映射的基本策略是首先将阴影的相关信息存储到阴影缓冲区中，然后正式绘制场景时根据阴影缓冲区的对应信息判断要绘制的片元是否处于阴影中，具体的步骤如图15-23所示。

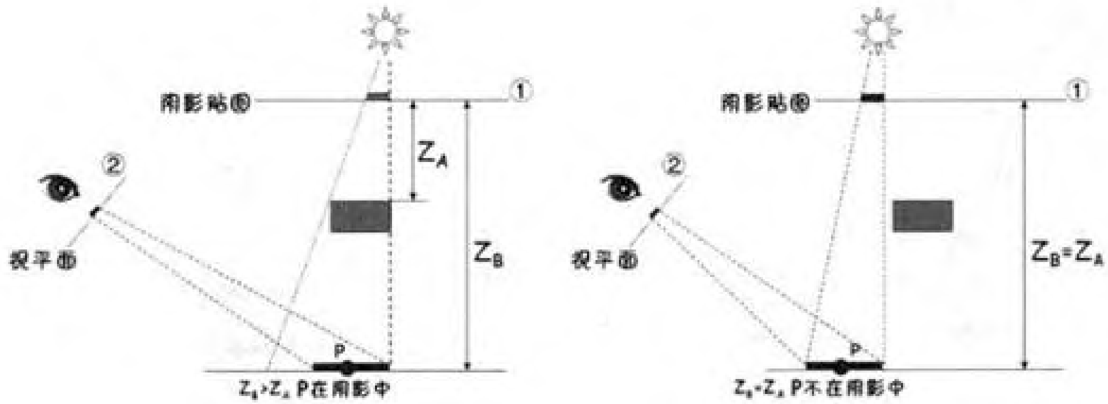


图15-23　阴影映射的具体步骤

图15-23中给出了阴影映射的基本步骤，具体情况如下所列。

- 首先将摄像机设置到光源所在的位置（观察方向面对场景，此时的摄像机观察矩阵称为观察矩阵1），同时设置合理的投影矩阵（这里称此矩阵为投影矩阵1）。
- 然后对场景进行第一次绘制，此次绘制将每个可以看见的片元（可以观察到指的是片元到光源间无遮挡）到光源的距离记录到一幅纹理图中的对应像素中，供后面的步骤使用。
- 接着将摄像机恢复到实际摄像机所处的位置，绘制场景。此次绘制时将前面步骤产生的纹理采用投影贴图的方式应用到场景中，进行投影贴图时采用的投影矩阵为投影矩阵1、观察矩阵为观察矩阵1。

• 绘制每个片元时，根据投影贴图纹理采样的结果换算出光源与此片元连线中距光源最近的片元距离（Z_A），再计算出此片元距光源的实际距离（Z_B）。若$Z_B > Z_A$，则需要绘制的片元处于阴影中，采用阴影的颜色着色，否则此片元不在阴影中，进行既定的光照着色。

提示

上述策略的关键在于产生记录距离的纹理图后投影贴图的使用，其使得需要绘制的片元能够通过纹理采样获取在自己与光源的连线上是否有其他片元遮挡的信息。

15.8.2 距离纹理图的生成

从上一小节的介绍中可以看出，阴影映射中第一项重要的工作是用于记录距离的纹理图的产生。而此纹理图的内容是通过将摄像机放置到光源位置观察场景绘制出来的，因此第一次绘制时需要将内容绘制到特殊的位置，而不是对应到屏幕的帧缓冲。这个特殊的位置可以采用自定义帧缓冲区，其结构如图15-24所示。

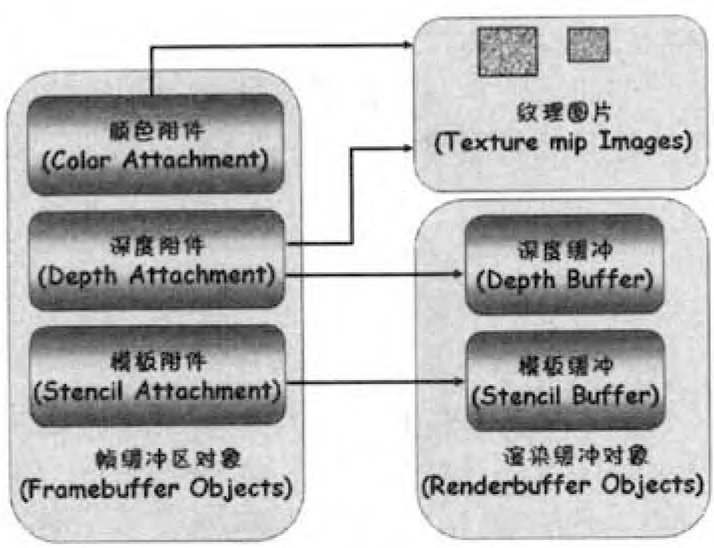


图15-24　自定义帧缓冲区的结构

从图15-24中可以看出，自定义帧缓冲包含3种可选的附件：颜色附件、深度附件、模板附件，具体情况如下所列。

- 颜色附件只能用纹理图来实现，本节案例中的距离信息就是记录在颜色附件对应的纹理图中的。
- 深度附件有两个选择，可以采用纹理图，也可以采用渲染缓冲对象（RenderbufferObject），本节案例采用的是渲染缓冲对象。
- 模板附件仅能采用渲染缓冲对象，用于进行模板测试。

了解了自定义帧缓冲区的基本情况后，就可以开发仅执行第一次绘制并将产生的纹理图贴到一个纹理矩形上进行显示的案例第一版Sample15_8_V1了，其运行效果如图15-25所示，具体的开发步骤如下所列。

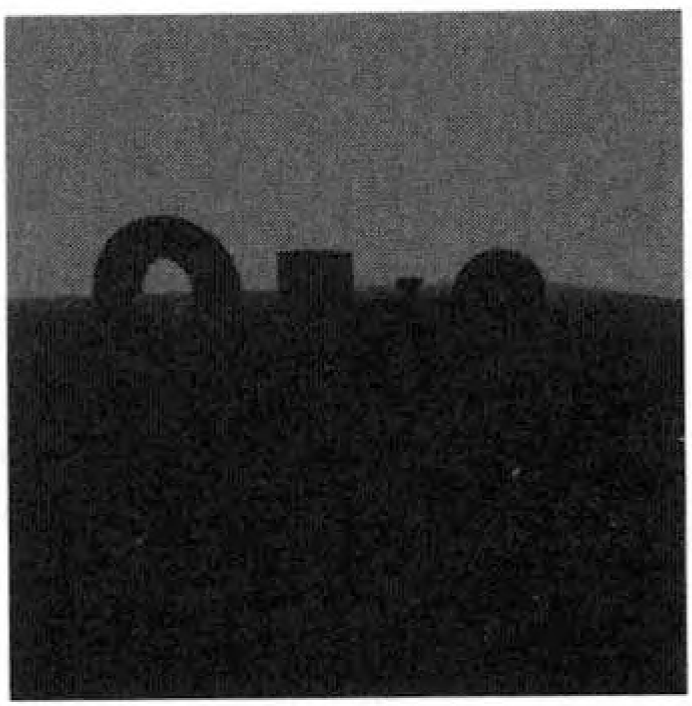


图15-25　记录距离的纹理图

（1）首先用3DMax生成5个基本物体（平面、圆环、茶壶、立方体、圆球），并导出成obj文件放入项目的assets目录待用。

（2）开发出搭建场景的基本代码，包括：加载物体、摆放物体、计算光照等。这些代码与前面章节的许多案例基本套路完全一致，因此这里不再赘述。

（3）由于进行距离纹理的绘制时会采用特殊的着色器，因此与此组着色器对应的初始化与调用 代码需要单独开发一套，补充进代表加载物体的LoadedObjectVertexNormal类，具体内容如下。

代码位置：见随书光盘中源代码/第15章/Sample15_8_V1/com/bn/Sample15_8_V1目录下的LoadedObjectVertexNormal.java。

```java
package com.bn.gles20.ex54;
……//此处省略了部分类的引入代码，读者可自行查看随书光盘中的源代码
public class LoadedObjectVertexNormal{//加载后的物体
  ……//此处省略了部分成员变量的声明，读者可自行查看随书光盘中的源代码
  int mProgramForShadow;                    //自定义着色器程序 id(距离纹理用)
  int muMVPMatrixHandleForShadow;           //总变换矩阵引用(距离纹理用)
  int muMMatrixHandleForShadow;             //位置、旋转、缩放变换矩阵引用(距离纹理用)
  int maPositionHandleForShadow;            //顶点位置属性引用(距离纹理用)
  int maLightLocationHandleForShadow;       //光源位置属性引用(距离纹理用)
      String mVertexShaderForShadow;        //顶点着色器
      String mFragmentShaderForShadow;      //片元着色器
     public LoadedObjectVertexNormal(MySurfaceView mv,float[] vertices,float[] normals) {
       initVertexData(vertices,normals);  //初始化顶点数据
       intShader(mv);                     //初始化正常绘制时的着色器
       initShaderForShadow(mv);           //初始化绘制距离纹理的着色器
     }
     public void initVertexData(float[] vertices,float[] normals)   {/*代码省略*/ }
     public void intShader(MySurfaceView mv)   {/*代码省略*/ }
     public void initShaderForShadow(MySurfaceView mv)    {//初始化绘制距离纹理的着色器
       mVertexShaderForShadow=ShaderUtil.loadFromAssetsFile( //加载顶点着色器的脚本内容
            "vertex_shadow.sh", mv.getResources());
       mFragmentShaderForShadow=ShaderUtil.loadFromAssetsFile(//加载片元着色器的脚本内容
            "frag_shadow.sh", mv.getResources());
       mProgramForShadow = ShaderUtil.createProgram( //基于顶点着色器与片元着色器创建程序
            mVertexShaderForShadow, mFragmentShaderForShadow);
       maPositionHandleForShadow = GLES20.glGetAttribLocation(//获取程序中顶点位置属性引用
            mProgramForShadow, "aPosition");
       muMVPMatrixHandleForShadow = GLES20.glGetUniformLocation(//获取程序中总变换矩阵引用
            mProgramForShadow, "uMVPMatrix");
       muMMatrixHandleForShadow = GLES20.glGetUniformLocation( //获取位置、旋转变换矩阵引用
            mProgramForShadow, "uMMatrix");
       maLightLocationHandleForShadow=GLES20.glGetUniformLocation(//获取程序中光源位置引用
            mProgramForShadow, "uLightLocation");
     }
     public void drawSelf(int texId,float[] mMVPMatrixGY)  {/*代码省略*/ }
     public void drawSelfForShadow() {                    //绘制物体的方法(距离纹理用)
       GLES20.glUseProgram(mProgramForShadow);//指定使用某套着色器程序
       GLES20.glUniformMatrix4fv(muMVPMatrixHandleForShadow, //将最终变换矩阵传入渲染管线
            1, false, MatrixState.getFinalMatrix(), 0);
       GLES20.glUniformMatrix4fv(muMMatrixHandleForShadow, //将基本变换矩阵传入渲染管线
            1, false, MatrixState.getMMatrix(), 0);
       GLES20.glUniform3fv(maLightLocationHandleForShadow,  //将光源位置传入渲染管线
            1, MatrixState.lightPositionFB);
       GLES20.glVertexAttribPointer (maPositionHandleForShadow, //将顶点位置传入渲染管线
            3, GLES20.GL_FLOAT, false,3*4, mVertexBuffer );
       GLES20.glEnableVertexAttribArray(maPositionHandleForShadow); //启用顶点位置数据
       GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount);         //绘制加载的物体
     }
}
```

说明

上述代码中主要新增了为距离纹理绘制功能服务的一些成员变量以及initShaderForShadow与drawSelfForShadow两个方法。initShaderForShadow方法负责初始化用于绘制距离纹理的着色器，drawSelfForShadow负责绘制距离纹理。这两个新增方法套路与一般绘制时的相同，只是传入管线的数据有所不同。

（4）完成了LoadedObjectVertexNormal类的修改后，就可以对渲染场景的MySurfaceView类进行修改了，修改后的代码框架如下。

代码位置：见随书光盘中源代码/第15章/Sample15_8_V1/com/bn/Sample15_8_V1目录下的MySurfaceView.java。

```
package com.bn.gles20.ex54;
……//此处省略了部分类的引入代码，读者可自行查看随书光盘中的源代码
class MySurfaceView extends GLSurfaceView {
    private final float TOUCH_SCALE_FACTOR = 180.0f/320;//角度缩放比例
   ……//此处省略了部分成员变量的声明，读者可自行查看随书光盘中的源代码
   float[] mMVPMatrixGY;//光源投影、观察组合矩阵
    public MySurfaceView(Context context) {/*代码省略*/ }
   @Override //触摸事件回调方法
   public boolean onTouchEvent(MotionEvent e) {/*代码省略*/ }
       private class SceneRenderer implements GLSurfaceView.Renderer {
          ……//此处省略了部分成员变量的声明，读者可自行查看随书光盘中的源代码
          TextureRect tr;                         //用于在屏幕上绘制距离纹理的纹理矩形
          int frameBufferId;                      //自定义帧缓冲 id
          int shadowId;                           //距离纹理的纹理 Id
          int renderDepthBufferId;                //用作深度缓冲的渲染缓冲对象
          boolean isBegin=true;                   //深度缓冲初始化完成标志
          public void initFRBuffers() {/*代码省略，后面进行介绍*/ }
          public void generateShadowImage(){/*代码省略，后面进行介绍*/ }
          public void drawShadowTexture() {       //将距离纹理呈现到屏幕上的方法
           GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, 0);
              GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT//清除深度缓冲与颜色缓冲
             | GLES20.GL_COLOR_BUFFER_BIT);
              GLES20.glClearColor(1f,0.3f,0.3f,1.0f);    //设置背景色
              MatrixState.setCamera(0,0,1.5f,0f,0f,-1f,0f,1.0f,0.0f);//设置摄像机
              MatrixState.setProjectOrtho(-0.6f,0.6f, -0.6f, 0.6f, 1, 10); //设置投影
              tr.drawSelf(shadowId);                      //绘制呈现距离纹理的纹理矩形
            GLES20.glDeleteFramebuffers(1, new int[]{frameBufferId}, 0);//删除自定义帧缓冲
              GLES20.glDeleteTextures(1, new int[]{shadowId}, 0);     //删除距离纹理
           }
           public void onDrawFrame(GL10 gl) {                 //在屏幕上绘制画面
                MatrixState.setLightLocation(lx, ly, lz);        //设置光源位置
```

```
            generateShadowImage();                    //通过绘制产生距离纹理
            drawShadowTexture();                      //绘制距离纹理到屏幕
        }
        public void onSurfaceChanged(GL10 gl, int width, int height)  {/*代码省略*/ }
        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
          ……//此处省略了部分代码，与前面很多案例中的相同
          tr=new TextureRect(MySurfaceView.this); //创建用于显示距离纹理的纹理矩形
}}}
```

说明

上述代码中增加了初始化自定义帧缓冲的initFRBuffers方法、通过绘制产生距离纹理的generateShadowImage方法以及将距离纹理呈现到屏幕上的drawShadowTexture方法。前两个方法将在后面的步骤中进行详细的介绍，drawShadowTexture与前面章节中绘制普通纹理图形的方法基本相同。

（5）完成了MySurfaceView类的代码框架后，下面将开发初始化自定义帧缓冲的initFRBuffers方法，具体代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_8_V1/com/bn/Sample15_8_V1目录下的MySurfaceView.java。

```
public void initFRBuffers()    {                     //初始化帧缓冲和渲染缓冲的方法
    int[] tia=new int[1];                           //用于存放产生的帧缓冲 id 的数组
    GLES20.glGenFramebuffers(1, tia, 0);            //产生一个帧缓冲 id
    frameBufferId=tia[0];                           //将帧缓冲 id 记录到成员变量中
    if(isBegin){                                    //若没有产生过深度渲染缓冲对象则产生一个
      GLES20.glGenRenderbuffers(1, tia, 0);         //产生一个渲染缓冲 id
      renderDepthBufferId=tia[0];                   //将渲染缓冲 id 记录到成员变量中
      GLES20.glBindRenderbuffer(GLES20.GL_RENDERBUFFER,
      renderDepthBufferId);                         //绑定指定 id 的渲染缓冲
          GLES20.glRenderbufferStorage(             //为渲染缓冲初始化存储
            GLES20.GL_RENDERBUFFER,
            GLES20.GL_DEPTH_COMPONENT16,            //内部格式为 16 位深度
            SHADOW_TEX_WIDTH,                       //缓冲宽度
            SHADOW_TEX_HEIGHT);                     //缓冲高度
            isBegin=false;                          //将未初始化标志设置为 false
        }
        int[] tempIds = new int[1];                 //用于存放产生纹理 id 的数组
        GLES20.glGenTextures(1, tempIds, 0);        //产生一个纹理 id
        shadowId=tempIds[0];                        //将纹理 id 记录到距离纹理 id 成员变量中
    }
```

- 第2-4行产生了自定义缓冲的id，并记录进成员变量已备后面的方法使用。
- 第5-16行初始化了用于实现深度缓冲的渲染缓冲对象，并为其初始化了存储。
- 第17-19行产生了距离纹理对应的纹理id并记录进成员变量以备后面的方法使用。

（6）完成了initFRBuffers方法的开发后，就可以开发通过绘制产生距离纹理的generateShadowImage方法了，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_8_V1/com/bn/Sample15_8_V1目录下的MySurfaceView.java。

```
1 public void generateShadowImage(){ //通过绘制产生阴影纹理
2     initFRBuffers();                    //初始化帧缓冲和渲染缓冲
3     GLES20.glViewport(0, 0, SHADOW_TEX_WIDTH, SHADOW_TEX_HEIGHT);     //设置视口
4     GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, frameBufferId);   //绑定帧缓冲
5     GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, shadowId);             //绑定纹理
6     ……//此处省略了设置纹理采样与拉伸方式的代码
7     GLES20.glFramebufferTexture2D(              //设置自定义帧缓冲的颜色附件
8       GLES20.GL_FRAMEBUFFER,
9       GLES20.GL_COLOR_ATTACHMENT0,              //颜色附件
10          GLES20.GL_TEXTURE_2D,                 //类型为2D纹理
11          shadowId,                             //纹理id
12          0                                     //层次
13        );
14        GLES20.glTexImage2D(                    //设置颜色附件纹理图的格式
15            GLES20.GL_TEXTURE_2D,
16            0,                                  //层次
17            GLES20.GL_RGB,                      //内部格式
18            SHADOW_TEX_WIDTH,                   //宽度
19            SHADOW_TEX_HEIGHT,                  //高度
20            0,                                  //边界宽度
21            GLES20.GL_RGB,                      //格式
22            GLES20.GL_UNSIGNED_SHORT_5_6_5,     //类型
23            null
24        );
25        GLES20.glFramebufferRenderbuffer(       //设置自定义帧缓冲的深度缓冲附件
26            GLES20.GL_FRAMEBUFFER,
27            GLES20.GL_DEPTH_ATTACHMENT,         //深度缓冲附件
28            GLES20.GL_RENDERBUFFER,             //渲染缓冲
29            renderDepthBufferId                 //渲染缓冲id
30        );
31       GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT//清除深度缓冲与颜色缓冲
32       | GLES20.GL_COLOR_BUFFER_BIT);
33       MatrixState.setCamera(lx,ly,lz,0f,0f,0f,0f,1,0);     //将摄像机设置到光源位置
34       MatrixState.setProjectFrustum(-1, 1, -1.0f, 1.0f, 1.5f, 400); //设置投影矩阵
35       mMVPMatrixGY=MatrixState.getViewProjMatrix();    //获取摄像机投影组合矩阵
36       lovo_pm.drawSelfForShadow();                     //绘制最下面的平面
37       MatrixState.pushMatrix();                        //保护现场
38       MatrixState.translate(-cDis, 0, 0);              //平移变换
39       lovo_qt.drawSelfForShadow();                     //绘制球体
40       MatrixState.popMatrix();                         //恢复现场
41       …//此处省略了绘制圆环、长方体、茶壶的代码，与绘制球体的基本相同
42   }
```

- 第4-30行主要为对自定义帧缓冲进行各方面设置的代码，首先将自定义帧缓冲的颜色附件设置为纹理图，然后对此纹理图的各方面进行设置，接着设置了自定义帧缓冲的深度缓冲附件。
- 第31-42行为向距离纹理中绘制各物体的代码，与前面案例中绘制物体的代码基本没有区别。

（7）完成了上述工作后，Java部分的开发就完成了，下面就应该进行着色器的开发了。本案例中包涵两套着色器，一套是用来进行距离纹理绘制的，另一套是进行普通纹理贴图的。进行普通纹理贴图的

在前面很多案例中都有使用，这里不再赘述。下面着重介绍进行距离纹理绘制的着色器，首先是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_8_V1/assets目录下的vertex_shadow.sh。

```
uniform mat4 uMVPMatrix;                    //总变换矩阵
uniform mat4 uMMatrix;                      //变换矩阵
attribute vec3 aPosition;                   //顶点位置
varying vec4 vPosition;                     //用于传递给片元着色器的顶点位置
void main() {
  gl_Position = uMVPMatrix*vec4(aPosition,1);    //根据总变换矩阵计算此次绘制此顶点的位置
  vPosition=uMMatrix*vec4(aPosition,1);          //计算出变换后的顶点位置并传递给片元着色器
}
```

> **说明**
>
> 上述顶点着色器的代码与前面很多案例的非常类似，主要工作是计算出顶点的绘制位置以及变换后位置。

（8）完成了顶点着色器的开发后，就可以开发片元着色器了，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_8_V1/assets目录下的frag_shadow.sh。

```
precision mediump float;            //给出默认的浮点精度
varying vec4 vPosition;             //接收从顶点着色器过来的顶点位置
uniform highp vec3 uLightLocation; //光源位置
void main(){
  float dis=distance(vPosition.xyz,uLightLocation);      //计算被照射片元到光源的距离
  float zsbf=floor(dis);                                  //取出距离的整数部分
  float xsbf=fract(dis);                                  //取出距离的小数部分
  xsbf=floor(xsbf*1024.0);                                //将小数部分放大1024倍取整
      float hzsbf=floor(zsbf/256.0);                      //将整数部分高位提取出来
      float lzsbf=mod(zsbf,256.0);                        //将整数部分低位提取出来
      float hxsbf=floor(xsbf/32.0);                       //将小数部分高位提取出来
      float lxsbf=mod(xsbf,32.0);                         //将小数部分低位提取出来
      float r=hzsbf/256.0;                                //将整数部分高位压缩到0-1
      float g=lzsbf/256.0;                                //将整数部分低位压缩到0-1
      float b=hxsbf/32.0;                                 //将小数部分高位压缩到0-1
      float a=lxsbf/32.0;                                 //将小数部分低位压缩到0-1
      gl_FragColor=vec4(r,g,b,a);              //将距离值的各部分存入片元的RGBA 4个色彩通道
   }
```

- 上述片元着色器很特殊，主要功能为将片元与光源的距离计算出来并拆解成4个部分再压缩存储到片元的RGBA 4个色彩通道中。当进行阴影绘制时就可以取出对应片元中各色彩通道的值并恢复成距离以帮助计算了。
- 可以想象，距离值的小数部分会重复出现（例如距离从0～10，那么就会有0.5、1.5、2.5 等多个值为0.5的小数部分），因此距离纹理中会有以光源为中心的同心圆条纹，主要是来自蓝色（B）通道。由于前面的插图是灰度印刷，可能会看不清楚，读者可以参考本书前面所付的彩图或自行运行案例观察。

15.8.3　阴影场景的绘制

完成了距离纹理的开发后，就可以进一步开发带阴影的场景了。主要工作为将上一小节的案例复制重命名为Sample15_8_V2，并增加了一些Java代码与着色器，步骤如下所列。

（1）在MySurfaceView类中增加绘制带阴影场景的drawScene方法，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_8_V2/com/bn/Sample15_8_V2目录下的MySurfaceView.java。

```
1 public void drawScene(GL10 gl)  {
2   GLES20.glViewport(0, 0, (int)SCREEN_WIDTH, (int)SCREEN_HEIGHT); //设置视口
3   GLES20.glBindFramebuffer(GLES20.GL_FRAMEBUFFER, 0);
4   //清除深度缓冲与颜色缓冲
5   GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT | GLES20.GL_COLOR_BUFFER_BIT);
6   MatrixState.setCamera(cx,cy,cz,0f,0f,0f,0f,1f,0f);//设置摄像机
7   MatrixState.setProjectFrustum(-ratio, ratio, -1.0f, 1.0f, 2, 1000);  //设置投影矩阵
8     ……此处省略了清除深度缓冲与颜色缓冲以及设置摄像机与投影矩阵的代码
9   lovo_pm.drawSelf(shadowId,mMVPMatrixGY);                    //绘制最下面的平面
10      MatrixState.pushMatrix();                               //保护现场
11      MatrixState.translate(-cDis, 0, 0);                     //平移变换
12      lovo_qt.drawSelf(shadowId,mMVPMatrixGY);                //绘制球体
13      MatrixState.popMatrix();                                //恢复现场
14      ……///此处省略了绘制圆环、长方体、茶壶的代码，与绘制球体的基本相同
15      GLES20.glDeleteFramebuffers(1, new int[]{frameBufferId}, 0);//删除自定义帧缓冲
16      GLES20.glDeleteTextures(1, new int[]{shadowId}, 0);     //删除距离纹理
17    }
```

- 第3行将帧缓冲绑定到0号缓冲（绘制到屏幕的缓冲，系统默认的）。
- 第4-14行为传统的绘制场景的代码，与前面章节案例中的没有太大的不同。
- 第15-16行在绘制完一帧画面后删除了自定义帧缓冲与距离纹理，这是因为每次绘制一帧时都会重新申请。若只申请，不删除，则程序运行一会儿后会内存崩溃。

（2）修改MySurfaceView类中绘制一帧画面的onDrawFrame方法，具体代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_8_V2/com/bn/Sample15_8_V2目录下的MySurfaceView.java。

```
public void onDrawFrame(GL10 gl){                //绘制一帧画面方法
  MatrixState.setLightLocation(lx, ly, lz);      //设置光源位置
  generateShadowImage();                         //通过绘制产生距离纹理
  drawScene(gl);                                 //绘制场景
}
```

说明

上述方法中仅仅是将绘制距离纹理到屏幕的drawShadowTexture方法替换成了绘制带阴影场景的drawScene方法。

（3）完成了Java代码的修改后，就可以开发着色器了。这里仅仅是对普通的光照着色器进行了修改，增加了投影贴图与阴影计算的相关代码。首先是顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_8_V2/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                    //总变换矩阵
uniform mat4 uMMatrix;                      //变换矩阵
uniform mat4 uMVPMatrixGY;                  //总变换矩阵(光源)
uniform vec3 uLightLocation;                //光源位置
uniform vec3 uCamera;                       //摄像机位置
attribute vec3 aPosition;                   //顶点位置
attribute vec3 aNormal;                     //顶点法向量
varying vec4 ambient;                       //用于传递给片元着色器的环境光最终强度
varying vec4 diffuse;                       //用于传递给片元着色器的散射光最终强度
varying vec4 specular;                      //用于传递给片元着色器镜面光最终强度
varying vec4 vPosition;                     //用于传递给片元着色器的顶点位置
……//此处省略了定位光光照计算的方法pointLight，与前面很多案例的相同
void main(){
  gl_Position = uMVPMatrix*vec4(aPosition,1);//根据总变换矩阵计算此次绘制此顶点的位置
  pointLight(normalize(aNormal),ambient,diffuse,specular,//计算光照各个通道的强度
        uLightLocation,vec4(0.1,0.1,0.1,1.0),vec4(0.7,0.7,0.7,1.0),vec4(0.3,0.
        3,0.3,1.0));
  vPosition=uMMatrix*vec4(aPosition,1);     //将变换后的顶点位置传递给片元着色器
}
```

说明

上述顶点着色器的代码与普通光照案例的基本相同，没有太大变化，主要就是增加了将变换后的顶点位置传递给片元着色器的代码。

（4）完成了顶点着色器后，就可以修改片元着色器了，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_8_V2/assets目录下的frag.sh。

```
precision mediump float;                       //给出默认浮点精度
uniform sampler2D sTexture;                    //纹理内容数据
uniform highp vec3 uLightLocation;             //光源位置
uniform highp mat4 uMVPMatrixGY;               //总变换矩阵(光源)
varying vec4 ambient;                          //接收从顶点着色器传来的环境光最终强度
varying vec4 diffuse;                          //接收从顶点着色器传来的散射光最终强度
varying vec4 specular;                         //接收从顶点着色器传来的镜面光最终强度
varying vec4 vPosition;                        //接收从顶点着色器传来的变换后顶点位置
void main(){
      //将片元的位置投影到光源处虚拟摄像机的近平面上
      vec4 gytyPosition=uMVPMatrixGY * vec4(vPosition.xyz,1);
      gytyPosition=gytyPosition/gytyPosition.w;     //进行透视除法
      float s=(gytyPosition.s+1.0)/2.0;             //将投影后的坐标换算为纹理坐标
      float t=(gytyPosition.t+1.0)/2.0;
      vec4 depth4=texture2D(sTexture, vec2(s,t));   //对投影纹理(距离纹理)图进行采样
      //将采样出的颜色值换算成最小距离值 ZA
      float minDis=depth4.r*256.0*256.0+depth4.g*256.0+depth4.b+depth4.a/32.0;
```

```
     float currDis=distance(vPosition.xyz,uLightLocation); //计算光源到此片元的距离 ZB
     vec4 finalColor=vec4(0.95,0.95,0.95,1.0);              //物体的颜色
     if(s>=0.0&&s<=1.0&&t>=0.0&&t<=1.0) {       //若纹理坐标在合法范围内则考虑投影贴图
          if(minDis<=currDis-3.0) {                  //若实际距离大于最小距离则在阴影中
              gl_FragColor= finalColor*ambient*1.2;  //仅用环境光着色
          } else{//不在阴影中用 3 个通道光照着色
              gl_FragColor = finalColor*ambient+finalColor*specular+finalColor
              *diffuse;
          }
     } else{    //不在阴影中用 3 个通道光照着色
         gl_FragColor = finalColor*ambient+finalColor*specular+finalColor*diffuse;
     }
  }
```

• 上述着色器最大的变化是首先要对距离纹理进行基于投影贴图技术的纹理采样，然后将采样出的颜色值恢复成距离值（Z_A），接着计算出实际距离值（Z_B），最后根据Z_A与Z_B的关系决定片元是否在阴影中。

• 比较Z_A与Z_B的大小时加入了一个修正值3.0，此值一般需要根据具体情况进行调整。如果不加入修正值，场景绘制出来可能导致产生很严重的“自身阴影”问题，如图15-26所示。

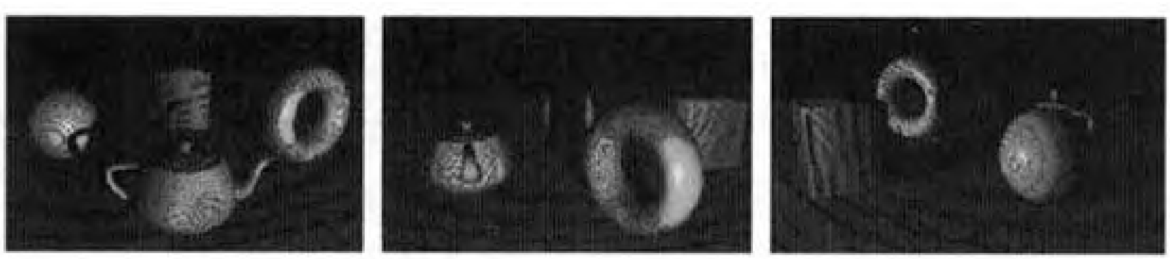


图15-26　自身阴影问题

到这里为止，基于阴影映射技术的阴影绘制就介绍完了，此技术比前一节中的平面阴影真实感有了较大的提升。但由于此技术中仅能将最近距离信息存储于纹理图中，精度不够高会产生自阴影现象，虽然通过修正值可以改善，但不能完全避免，使用时需要注意。

15.9　光线跟踪

前面两个小节分别介绍了平面阴影以及阴影映射技术，在效果要求不是极高的情况下这两种技术已经能够满足需要。但由于平面阴影只能应用于目标平面、阴影映射由于计算精度有自阴影的问题，故如果对效果要求很高的话，就难以满足要求了。

因此，本节将向读者简单介绍一种用于模拟逼真光学效果的技术——光线跟踪，其不但能模拟出十分逼真的光照阴影效果，也可以逼真地模拟折射、反射等光学效果。本节将通过一个具体的案例Sample15_9向读者介绍OpenGL ES 2.0中光线跟踪的实现。

15.9.1　案例效果与基本原理

介绍本案例的具体开发步骤之前首先需要了解一下案例的运行效果与基本原理，其运行效果如 图15-27所示。

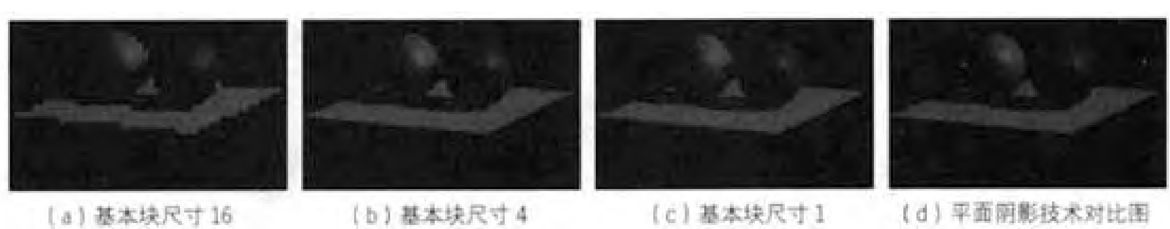


图15-27　光线跟踪案例运行效果图

说明

图15-27中图a、图b、图c分别为采用光线跟踪技术渲染场景时将基本块的尺寸设为16、4、1时的运行效果图。图（d）为采用前面15.7节介绍的平面阴影技术渲染同样场景的效果图，用于进行对比。

从图15-27中图c中可以看出，本节案例场景中放置了两个球体和一个平面。同时可以看出，球体与球体之间、球体与平面之间的阴影效果十分逼真。通过与图d进行比较可以明显地看出，光线跟踪所产生的阴影是十分真实的，没有平面阴影的固有缺陷。

提示

由于书中的插图采用灰度印刷可能看起来不是很清楚，此时请读者参照前面的彩页或自行采用真机运行观察。另外，由于光线跟踪算法的计算量较大，因此运行本案例时需要耐心等待一段时间（一般为60~120s），才可以看到案例的运行效果。这本身也是光线跟踪现阶段没有被广泛采用的主要障碍，以后随着硬件性能的逐步提升光线跟踪将有越来越多的用武之地。

虽然采用光线跟踪技术渲染出的场景与前面章节介绍的光照、阴影相关技术渲染出的场景看起来基本相同，但两者的渲染思路是截然不同的。

- 通过前面章节介绍的光照、阴影相关技术渲染场景时，是将场景中的物体经过各种变换后投影到近平面上再光栅化而呈现到屏幕上的。
- 通过光线跟踪技术渲染场景的基本思想为：通过摄像机和屏幕上一个基本块的位置确定一条光线，根据此光线与场景中物体的第一个碰撞点P的信息来确定屏幕上对应基本块的颜色值，其基本原理如图15-28所示。

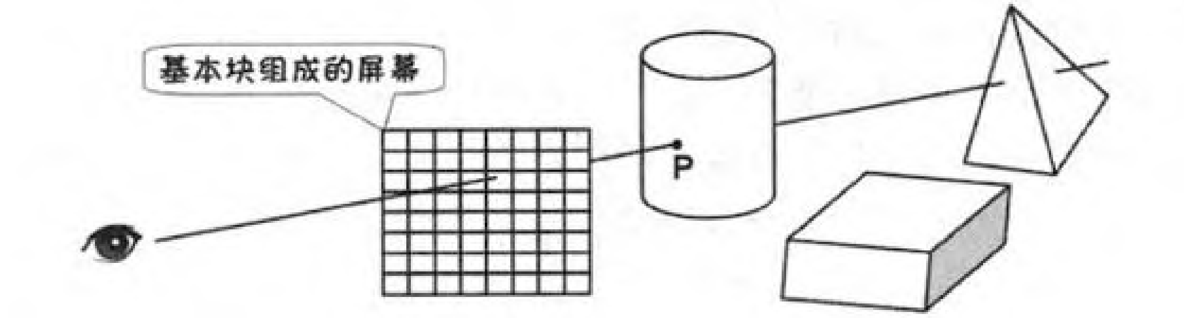


图15-28　光线跟踪的基本原理

实现光线跟踪时的几个技术要点如下所列。

- 将屏幕分成多个基本块，每个基本块可以由一个或多个像素组成。
- 将摄像机放置在适当的位置，然后遍历所有基本块，并通过摄像机的位置和基本块左下角的位置确定一条光线。
- 找出光线与场景中物体的第一个碰撞点，该碰撞点的信息将用来确定对应基本块的颜色。
- 判断第一交点是否在阴影中，以决定此基本块的最终颜色。

下面对每一个技术要点进行简要介绍，内容如下。

1．基本块的绘制

假设屏幕的物理分辨率为16×32，如果设置基本块的大小为4，那么每个基本块是由4×4个像素组成。由于同一基本块中所有像素的颜色是相同的，因此对每一个基本块进行光线跟踪后得到的渲染图像的实际分辨率为4×8，如图15-29所示。

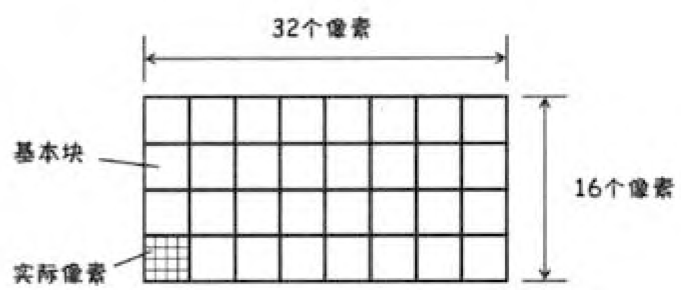


图15-29　基本块示意图

读者可能会有这样的疑问：为什么不直接固定采用1×1的渲染块呢，那样实际分辨率不就不会低于屏幕的物理分辨率了吗？这主要是考虑到性能的问题，基本块的尺寸大了虽然损失了一些效果，但可以大大提高渲染速度。因此实际开发中经常需要根据机器的硬件性能调整基本块的大小，以取得性能和效果的均衡。

2．通过摄像机与基本块确定光线

本书案例中采用的策略是通过摄像机位置及当前被考察的基本块左下角的位置确定光线方向的。一条光线r由起点位置start和传播方向dir组成，其可以表示成关于时间t的如下表达式。

r（t）=start+dir×t （其中t表示光线传播的时间）

由于光线跟踪技术中假设光是从摄像机发出的，因此光线的起点就是摄像机的位置eye，而光线的方向dir由摄像机的位置和基本块所在的位置来确定，表达式如下：

$$dir=-Nn+W(2c/nCols-1)u+H(2r/nRows-1)v$$

- N为摄像机与屏幕（近平面）的距离，W、H分别为近平面宽和高的一半。
- nCols,nRows分别为屏幕上像素的总列数和总行数。
- c、r分别为屏幕上基本块的列数和行数。
- u、v、n分别为摄像机坐标系3个轴的单位向量i、j、k在世界坐标系中的值，此3个值可以由Matrix类中的setLookAtM方法求出。

3．求出光线与场景中物体的第一个碰撞点

确定了通过各个基本块的各条光线后，下面就应该求出光线与场景中物体的第一个碰撞点了。由于此时光线的参数方程已知，因此只需要求出光线与第一个碰撞点的碰撞时间t_{hit}，然后将t_{hit}代入参数方程中，就可以求出碰撞点的位置，公式如下。

$$P_{hit}=eye+dir\times t_{hit}$$

> **提示**
>
> 求出了碰撞点的位置后，碰撞点的其他信息也就可以方便地求出了。如碰撞点的法向量、碰撞点所属物体表面处的颜色、碰撞点是否在阴影中等信息，其中判断碰撞点是否在阴影中的方法将在后面进行介绍。

接着要讨论的是，如何求出一条光线自发射后与场景中物体第一个碰撞点的碰撞时间t_{hit}。里以球体为例，首先介绍求光线与通用球面交点的算法。所谓通用球面是指半径为1、几何中心在原点的球面。为了简化公式，这里用S来表示eye，用c表示dir，那么光线的参数方程可表示为：

$$r(t)=S+ct$$

而通用球面的隐式表达式为：

$$F（x，y,z）= x^2 +y^2 +z^2 -1（或F（P）=|P|^2 -1)$$

球面上的任意一点满足F（P）=0。如果光线与球面在T时刻相交，则交点为：

$$r（T）=S +cT$$

将该交点坐标代入球面方程得到：

$$F（S +cT）=即|S +ct|^2 -1= 0$$

展开得到：

$$|c|^2 t^2 +2(S \bullet c)t +|S|^2 -1)=0$$

上述表达式可以看作t的二次等式：$At_2 +2Bt +C =0$ 其中A、B和C分别为：

$$A=|C|^2 \quad B=S \bullet c \quad C=|S|^2 -1$$

然后用一元二次方程的求根公式求解上述等式即可得到T的表达式：

$$T=-\frac{B}{A}\pm\frac{\sqrt{B^2-AC}}{A}$$

关于判别式 $B^2 -AC$ 需要分下列3种情况考虑。

- 如果判别式 $B^2 -AC$ 的值为负，则光线与球没有交点。
- 如果判别式 $B^2 -AC$ 的值为零，那么光线在球面上的某一点擦过球面，碰触到球面的时间为-B/A。
- 如果判别式的值为正，那么光线在两个时间点tl、t2和与球面相交，tl、t2分别对应上式中的“+”和“一”两种情况。

求出相交的时间后，将其代入光线的参数方程，就得到了交点的坐标。当有多个交点时可以根据时间t的大小来判断交点的先后位置。

当然，构造场景时不可能采用的都是类似于通用球面的通用物体（几何中心位于坐标原点的物体），物体应该可以以任何姿态位于场景中的任意位置。此时，设计者往往需要对场景中的每个物体施加一个仿射变换T，来确定物体在场景中的大小，姿态和位置。

但仿射变换后物体的隐式表达式会变得非常复杂，因此一般不采用计算光线与变换后物体交点的方法。而是采用类似于前面凹凸映射的处理思路，通过仿射变换T的逆变换T’，将光线变换到物体空间中，然后再求碰撞时间 t_{hit} ，这样就容易得多了。

因此求解实际交点坐标可以分为以下3个步骤。

- 求出逆变换后的光线$S' + c' t$。
- 求出逆变换后的光线与仿射变换前通用物体的碰撞时t_{hit}。
- 把碰撞时间t_{hit}代入等式变换前的光线$S + ct$得到实际的交点坐标。

设实施仿射变换T的矩阵为M，M^{-1}为M的逆矩阵。则逆变换后光线$S' + c' t$中的S'和c'可以由下面两个表达式求出。

$$\begin{pmatrix} S'_x \\ S'_y \\ S'_z \\ 1 \end{pmatrix} = M^{-1} \begin{pmatrix} S_x \\ S_y \\ S_z \\ 1 \end{pmatrix} \qquad \begin{pmatrix} c'_x \\ c'_y \\ c'_z \\ 0 \end{pmatrix} = M^{-1} \begin{pmatrix} c_x \\ c_y \\ c_z \\ 0 \end{pmatrix}$$

得到光线与仿射变换后物体的交点坐标后，接下来的问题是如何获得变换后交点处的法向量。通用物体表面的法向量m很容易求出，而计算出通用物体在交点处的法向量后，再乘以变换矩阵M的逆转置矩阵M^{-T}就可以求得变换后的法向量m’：

$m' = M^{-T} m$　（其中M^{-T}是变换矩阵M的逆转置矩阵）

4．判断第一交点是否在阴影中

判断交点是否处于阴影中的方法是，使用一条称为阴影探测器的新光线。此光线的起点为碰撞点向摄像机方向移动一个微小距离后的位置，其方向由光源位置与碰撞点位置决定，故阴影探测器的参数表达式如下：

$(P_{hit} - \varepsilon\, dir) + (L - P_{hit})\, t$　（其中L代表光源的位置）

> **说明**
>
> 其中p_{hit}为碰撞点的位置，ε是一个很小的正数，dir是摄像机发出的光线的方向，L是光源位置，t是传播时间。

从上述表达式中可以看出：当t=0时表达式表示的点接近于碰撞点处；当t=1时表达式表示的点接近于光源点处；当t在0到1之间时表达

式表示的点位于碰撞点和光源之间。计算碰撞点是否位于阴影中的策略也很简单，只需要将场景中的所有物体与阴影探测器对应的光线求交点，若有任何一个交点对应的t在0～1内则碰撞点在阴影中，否则碰撞点不在阴影中，如图15-30所示。

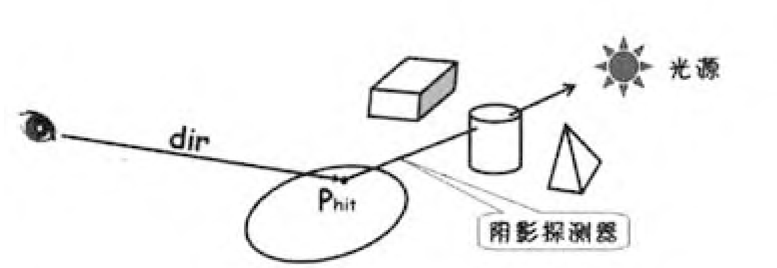


图15-30　判断物体上的点P_{hit} 是否处于阴影中

判断出碰撞点是否在阴影中后可以决定采用什么样的颜色对其对应的基本块进行着色，若在阴影中则仅采用环境光着色，若不在阴影中则同时要考虑环境光、散射光、镜面光。

15.9.2　开发步骤

上一小节介绍了光线跟踪的基本原理及案例的运行效果，本小节将介绍如何进行本节案例的开发，具体步骤如下。

提示

下面的步骤中仅给出了与光线跟踪技术密切相关的部分，与光线跟踪技术关系不密切，前面很多章节已经介绍过的内容这里就不再赘述了，需要的读者请自行参考随书光盘。

（1）首先开发的是表示基本块的ColorRect类，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_9/com/bn/Sample15_9目录下的ColorRect.java。

```java
package com.bn.Sample15_9;                          //声明包
……//此处省略了一些引入相关类的代码，需要的读者请参考光盘
public class ColorRect {
    int mProgram;                                   //自定义渲染管线着色器程序 id
    int muMVPMatrixHandle;                          //总变换矩阵引用
    int maPositionHandle;                           //顶点位置属性引用
    int muColorHandle;                              //片元颜色属性引用
    int mu3DPosHandle;                              //基本块对应的第一碰撞点位置一致变量引用
    int muNormalHandle;                             //基本块对应的第一碰撞点法向量一致变量引用
        int muLightLocationHandle;                  //光源位置一致变量引用
        int muCameraHandle;                         //摄像机位置一致变量引用
        int muIsShadow;                             //是否绘制阴影一致变量引用
        String mVertexShader;                       //顶点着色器脚本字符串
        String mFragmentShader;                     //片元着色器脚本字符串
        FloatBuffer   mVertexBuffer;                //顶点坐标数据缓冲
        int vCount=0;                               //顶点数量
        float[] color3 = new float[3];              //基本块对应的第一碰撞点的颜色
        float[] vertexPos3D = new float[3];         //基本块对应的第一碰撞点的位置
        float[] normal3D = new float[3];            //基本块对应的第一碰撞点的法向量
        float[] lightPos3D = new float[3];          //光源位置
        float[] cameraPos3D = new float[3];         //摄像机位置
        int isShadow;                               //基本块对应的第一碰撞点是否在阴影中标志
        float u;                                    //基本块在屏幕上的位置
        float v;
        public ColorRect(MySurfaceView mv){
         initVertexData();                          //初始化顶点坐标与着色数据
         intShader(mv);                             //初始化着色器
        }
        public void initVertexData(){/*代码省略*/}
        public void intShader(MySurfaceView mv){/*代码省略*/}
         public void drawSelf() {                   //绘制基本块的方法
             MatrixState.pushMatrix();              //保护现场
             MatrixState.translate(u, v, 0);        //移动到视口中的 u，v 位置处
             GLES20.glUseProgram(mProgram);         //指定使用某套着色器程序
             GLES20.glUniformMatrix4fv(             //将最终变换矩阵传入着色器程序
                     muMVPMatrixHandle, 1, false,MatrixState.getFinalMatrix(), 0);
             //将基本块对应的第一碰撞点颜色传入渲染管线
             GLES20.glUniform3fv(muColorHandle, 1, color3, 0);
             ……//此处省略了部分将数据传入渲染管线的代码，读者可自行查看随书光盘
             GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount);//绘制矩形
             MatrixState.popMatrix();               //恢复现场
        }
       public void setColor(float r,float g,float b){         //设置颜色的方法
        this.color3[0] = r; this.color3[1] = g; this.color3[2] = b;
       }
        ……//此处省略了将部分设置其他属性的方法，读者可自行查看随书光盘中的源代码
  }
```

说明

每一个ColorRect类的对象表示一个基本块，也就是说每个ColorRect类的对象实际上就是一个颜色正方形，其边长就是基本块的尺寸。每个基本块的颜色由光线跟踪算法根据场景情况计算出来，并通过setColor方法传递给基本块。

（2）开发完表示基本块的ColorRect类后，接着就可以开发渲染基本块的片元着色器了，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_9/assets目录下的frag.sh。

```
precision mediump float;                              //给出默认的浮点精度
uniform vec3 uColor;                                  //基本块的颜色
uniform vec3 uPosition;                               //基本块对应的第一碰撞点位置
uniform vec3 uNormal;                                 //基本块对应的第一碰撞点法向量
uniform vec3 uLightLocation;                          //光源位置
uniform vec3 uCamera;                                 //摄像机位置
uniform int isShadow;                                 //阴影绘制标志
……//此处省略了计算定位光光照的pointLight方法，读者可自行查看随书光盘中的源代码
void main(){
      vec4 baseColor = vec4(uColor,1.0);              //基本颜色
      if(isShadow == 0){                              //若不在阴影中则需要计算光照
            vec4 ambient, diffuse, specular;
            pointLight(ambient,diffuse,specular,
 vec4(0.15,0.15,0.15,1.0),vec4(0.9,0.9,0.9,1.0),vec4(0.7,0.7,0.7,1.0));
            //综合3个通道光的最终强度及片元的颜色计算出最终片元的颜色并传递给管线
            gl_FragColor=baseColor*ambient + baseColor*diffuse + baseColor*specular;
      } else {                                        //如果在阴影中只计算环境光
            gl_FragColor=baseColor*vec4(0.15,0.15,0.15,1.0);
      }
   }
```

说明

此片元着色器的主要功能是根据管线传入的基本块对应的第一碰撞点的位置、法向量、颜色以及光源位置、摄像机位置进行光照计算以得到基本块的最终颜色。

（3）为了代码开发与维护的方便，接着需要开发的是用于统一管理程序中各个常量的Constant类，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_9/com/bn/Sample15_9目录下的Constant.java。

```
1 package com.bn.Sample15_9;                                     //声明包
2 public class Constant {
3     //关于屏幕尺寸的常量
4     public static final float SCREEN_WIDTH = 800;               //屏幕的宽度
5     public static final float SCREEN_HEIGHT = 480;              //屏幕的高度
6     //关于渲染时的常量
7     public static final float blockSize = 4f;                   //基本块的尺寸
8     public static final float W = SCREEN_WIDTH / 2.0f;          //屏幕半宽
9     public static final float H = SCREEN_HEIGHT / 2.0f;         //屏幕半高
10        public static final float ratio = W / H;                //屏幕宽高比
11        public static final float nRows = SCREEN_HEIGHT;        //屏幕像素总行数
12        public static final float nCols = SCREEN_WIDTH;         //屏幕像素总列数
13        //关于真实3D世界中近平面的量
14        public static final float N_3D = 24;                    //近平面到摄像机的距离
15        public static final float W_3D = ratio;                 //近平面半宽
16        public static final float H_3D = 1.0f;                  //近平面半高
17        //关于真实世界中各物体的量
18        public static final float R = 0.6f;                     //球的半径
19        public static final float CENTER_DIS = 0.7f;     //球心与世界坐标系Y轴的距离
20        public static final float PLANE_WIDTH = 3.5f;                 //平面宽度
21        public static final float PLANE_HEIGHT = 4f;                  //平面长度
22        public static final float[] BALL1_COLOR = {0.8f,0.2f,0.2f};   //球1的颜色
23        public static final float[] BALL2_COLOR = {0.2f,0.2f,0.8f};   //球2的颜色
24        public static final float[] PLANE_COLOR = {0.2f,0.8f,0.2f};   //平面的颜色
25        //关于摄像机的参数
26        public static final float CAM_X = 15;                   //摄像机位置X坐标
27        public static final float CAM_Y = 7;                    //摄像机位置Y坐标
28        public static final float CAM_Z = 32;                   //摄像机位置Z坐标
29        //关于光源的参数
30        public static final float LIGHT_X = 100;                //光源位置X坐标
31        public static final float LIGHT_Y = 80;                 //光源位置Y坐标
32        public static final float LIGHT_Z = 0;                  //光源位置Z坐标
33        //计算阴影时用的极小的正数e
34        public static final float MNIMUM = 0.00001f;
35    }
```

提示

此Constant类中给出了进行光线跟踪计算时需要用到的很多与场景相关的常量，读者可以根据需要进行调整。如第7行基本块的

尺寸，若希望得到最为细腻的画面则应该将其调整为1，若希望得到更快的渲染速度可以将其调整为8。

（4）接着需要开发的是几个方便编程的辅助类，包括代表向量的Vector3类、代表浮点数三元组的Point3类以及代表颜色的Color3f类。这几个类的实现都非常简单，需要的读者请参考随书光盘中的源代码。

（5）完成了几个辅助类的开发后，下面要做的工作是在MySurfaceView类中创建与实施光线跟踪相关的对象，具体代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_9/com/bn/Sample15_9目录下的MySurfaceView.java。

```
rect = new ColorRect(MySurfaceView.this);                          //创建用于绘制基本块的对象
light = new Light(new Point3(LIGHT_X,LIGHT_Y,LIGHT_Z));  //创建光源对象
cam=new Camera(light);                                                //创建摄像机对象
scn=new Scene(cam, light);                                            //创建场景对象
```

提示

光源、摄像机和场景类还没有开发，读者不用担心，后面步骤中会对各个类的开发进行介绍。

（6）在MySurfaceView类中创建了各个对象后，就需要对OpenGL ES中的摄像机、投影进行设置，还需要对光线跟踪中的摄像机、场景进行设置，并调用光线跟踪中摄像机对象的raytrace方法进行光线跟踪计算了，具体代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_9/com/bn/Sample15_9目录下的MySurfaceView.java。

```
MatrixState.setProjectOrtho(-W, W, -H, H, 1, 2);      //调用此方法计算产生正交投影矩阵
MatrixState.setCamera(0, 0, 1, 0, 0, 0, 0, 1, 0);      //设置绘制基本块时的camera位置
//设置光线跟踪算法中的真实摄像机位置
cam.setMyCamera(CAM_X, CAM_Y, CAM_Z, 0f, 0f, 0f, 0f, 1.0f, 0.0f);
scn.transform();                                                  //对场景中物体进行变换
cam.raytrace(scn, rect);                                         //开始光线跟踪以渲染物体
```

（7）完成了MySurfaceView类中与光线跟踪修改的工作后，接着介绍光线跟踪中的摄像机类Camera，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_9/com/bn/Sample15_9目录下的Camera.java。

```
package com.bn.Sample15_9;                    //声明包
……//此处省略了部分引入相关类的代码，需要的读者请参考随书光盘
public class Camera {
    Point3 eye,look,up;                       //摄像机位置、目标点和 UP 向量
    Vector3 u,v,n;                            //摄像机坐标系 3 个轴上的单位向量在世界坐标系中的值
    Light light;                              //光源位置
    public Camera(Light light){
        this.light = light;                   //初始化光源位置
        eye=new Point3(); look=new Point3(); up=new Point3();    //初始化摄像机 9 参数
            u=new Vector3(); v=new Vector3(); n=new Vector3();
        }
        private float[] vMatrix = new float[16];             //摄像机位置朝向 9 参数矩阵

        public void setMyCamera(//设置摄像机位置的方法，主要是为了计算 u、v、n 3 个向量的值
            float cx, float cy, float cz,                    //摄像机位置
            float tx, float ty, float tz,                    //摄像机目标点
            float upx, float upy, float upz                  //摄像机 UP 向量
        ){
            Matrix.setLookAtM(
                    vMatrix,                                 //结果矩阵
                    0,                                       //存放于矩阵中的起始位置
```

```
22                  cx, cy, cz,                //摄像机位置
23                  tx, ty, tz,                //摄像机目标点
24                  upx, upy, upz              //摄像机 UP 向量
25              );
26              eye.x=cx; eye.y=cy; eye.z=cz;      //初始化摄像机位置
27              look.x=tx; look.y=ty; look.z=tz;   //初始化目标点位置
28              up.x=upx; up.y=upy; up.z=upz;      //初始化 up 向量的 3 个分量
29              //从矩阵中取出 u、v、n 3 个向量的值，注意 OpenGL ES 中的向量按列存储
30              u.x=vMatrix[0]; u.y=vMatrix[4]; u.z=vMatrix[8];
31              v.x=vMatrix[1]; v.y=vMatrix[5]; v.z=vMatrix[9];
32              n.x=vMatrix[2]; n.y=vMatrix[6]; n.z=vMatrix[10];
33          }
34          public void raytrace(Scene scn, ColorRect rect) {    //光线跟踪的计算方法
35              System.out.println("start...");                  //打印开始时间和开始标志
36              long start = System.currentTimeMillis();
37              Ray theRay = new Ray();                          //创建光线对象
38              theRay.setStart(eye);                            //光线从眼睛开始
39              for (int col = 0; col < nCols; col += blockSize) {
                                                   //循环对每个基本块进行光线跟踪计算
40                for (int row = 0; row < nRows; row += blockSize) {
41                  //根据基本块所在行列数，计算光线方向向量 dir
42                  Vector3 dir1 = n.multiConst(-N_3D);
43                  Vector3 dir2 = u.multiConst(W_3D*(2*col/nCols-1));
44                  Vector3 dir3 = v.multiConst(H_3D*(2*row/nRows-1));
45                  Vector3 dir = dir1.add(dir2).add(dir3);
46                  theRay.setDir(dir);               //设置光线方向
47                  Color3f clr = new Color3f();      //创建用于存放第一个碰撞点颜色的对象
48                  Point3 vertexPos = new Point3();  //创建用于存放第一个碰撞点位置的对象
49                  Vector3 normal = new Vector3();   //创建用于存放第一个碰撞点法向量的对象
50                  //计算此光线对应的各个量的值
51                  int isShadowFlag = scn.shade(theRay, clr, vertexPos, normal);
52                  //如果光线和物体没有交点，说明是背景色，不进行绘制，继续下一条光线
53                  if(isShadowFlag == -1){ continue; }
54                  rect.setColor(clr.red, clr.green, clr.blue);     //第一碰撞点颜色
55                  rect.setPos3D(vertexPos.x, vertexPos.y, vertexPos.z);//第一碰撞点位置
56                  rect.setNormal3D(normal.x, normal.y, normal.z); //第一碰撞点法向量
57                  rect.setLightPos3D(light.pos.x, light.pos.y, light.pos.z);//光源位置
58                  rect.setCameraPos3D(eye.x, eye.y, eye.z);        //摄像机位置
59                  rect.setShadow(isShadowFlag);                    //是否在阴影中
60                  rect.setColRow(col, row);             //设置基本块在哪一行，哪一列
61                  rect.drawSelf();                      //绘制基本块
62              }}
63              long end = System.currentTimeMillis();    //打印结束时间和结束标志
64              System.out.println("time="+(end-start)/1000.0+"s");
65              System.out.println("finish...");
66          }
67      }
```

• 第4-6行为该类的成员变量，包括摄像机位置、目标点和UP向量，摄像机坐标系3个轴上 的单位向量在世界坐标系中的值，以及光源位置。

• 第14-33行为设置光线跟踪中摄像机位置的setMyCamera方法，该方法的主要功能是通过调用Matrix类中的setLookAtM方法生成摄像

机观察矩阵，并从矩阵中提取出u、v、n 3个向量的值。

- 第34-66行为光线跟踪的计算方法raytrace，在该方法中首先遍历所有的基本块，计算出每个基本块对应的光线方向。然后调用场景对象的shade方法对光线进行跟踪，找到第一个碰撞点的各项信息后存储到各个对象中，接着将各项信息传递给执行绘制的基本块对象，最后进行绘制。

（8）接着给出的是存储一个碰撞点各项信息的HitInfo类，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_9/com/bn/Sample15_9目录下的HitInfo.java。

```
package com.bn.Sample15_9;                              //声明包
public class HitInfo{
    double hitTime;                                     //相交时间
    HitObject hitObject;                                //相交的物体
    boolean isEntering;                                 //光线是进入还是射出
    int surface;                                        //相交于哪个表面
    Point3 hitPoint;                                    //交点的坐标，变换后的
    Vector3 hitNormal;                                  //交点处的法向量，变换前的
    public HitInfo(){
            hitPoint = new Point3();                    //创建碰撞点对象
            hitNormal = new Vector3();                  //创建法向量对象
        }
        public void set(HitInfo hit){                   //根据传入的HitInfo对象
            this.hitTime=hit.hitTime;                   //复制各项信息进入自身
            this.hitObject=hit.hitObject;               //的成员变量
            this.isEntering=hit.isEntering;
            this.surface=hit.surface;
            this.hitPoint.set(hit.hitPoint);
            this.hitNormal.set(hit.hitNormal);
        }
    }
```

说明

此类非常简单，主要是有一些成员变量用来记录碰撞点的各项信息。

（9）下面给出的是用于记录一条光线与场景中物体多个交点信息的类Intersection，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_9/com/bn/Sample15_9目录下的Intersection.java。

```
package com.bn.Sample15_9;                              //声明包
public class Intersection {
    int numHits;                                        //有正相交时间的相交物体的数目
    HitInfo[] hit=new HitInfo[8];                       //交点列表数组
    public Intersection(){
        for(int i=0; i<8; i++){                         //初始化交点列表数组
            hit[i]=new HitInfo();
        }
    }
        public void set(Intersection inter){            //将传入 Intersection 对象的信息
            for(int i=0; i<8; i++){                             //复制到自身
                this.hit[i].set(inter.hit[i]);
            }
            this.numHits = inter.numHits;
        }
}
```

说明

此类非常简单，主要是提供了一个数组用于存储一条光线与场景中物体的交点列表。

（10）接着需要开发的是表示光线跟踪中光线的Ray类，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_9/com/bn/Sample15_9目录下的Ray.java。

```
package com.bn.Sample15_9;                                  //声明包
public class Ray {
    Point3 start;                                           //光线起始位置
    Vector3 dir;                                            //光线方向
    public Ray(){
        start = new Point3(); dir = new Vector3();          //初始化光线起始位置及方向
    }
    public void setStart(Point3 start){                     //设置光线起始位置的方法
        this.start.x = start.x; this.start.y = start.y; this.start.z = start.z;
        }
        public void setDir(Vector3 dir){                    //设置光线发射方向的方法
            this.dir.x = dir.x; this.dir.y = dir.y; this.dir.z = dir.z;
        }
    }
```

说明

该类非常简单，主要是提供了两个成员变量用于记录光线的起始位置与方向。

（11）下面需要开发的是表示光源的Light类了，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_9/com/bn/Sample15_9目录下的Light.java。

```
package com.bn.Sample15_9;                                  //声明包
public class Light {
    Point3 pos;                                             //灯光位置
    public Light(){                                         //构造器 1
        pos = new Point3();
    }
    public Light(Point3 pos) {                              //构造器 2
        this.pos = pos;
    }
}
```

提示

不直接使用Point3表示光源而是封装成Light类是为了理解方便。

（12）接下来开发的是抽象类HitObject，场景中的任何物体类均继承自该类，其具体代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_9/com/bn/Sample15_9目录下的HitObject.java。

```
package com.bn.Sample15_9;                              //声明包
```

```java
import android.opengl.Matrix;                               //引入相关类
public abstract class HitObject {
    Color3f color;                                          //物体的颜色
    private float[] myMatrix;                               //变换矩阵
    Camera cam;                                             //摄像机
    public abstract boolean hit(Ray ray,Intersection inter); //求交点的方法
    public abstract boolean hit(Ray ray);                          //只判断是否相交的方法
    public Color3f getColor() {                                    //获取物体颜色的方法
            return color;
        }
        public Point3 rayPos(Ray r,double t){ //返回光线在 t 时刻的位置的方法(使用变换前的光线)
            return cam.eye.addVec(r.dir.multiConst((float)t));
                                                            //eye+dir*t 得到当前点的坐标
        }
        //获取变换后的光线(将 r 按逆变换矩阵变换后赋值给 genRay,(左乘变换矩阵的逆矩阵))
        public void xfrmRay(Ray genRay, float[] invTransf, Ray r){
            //求变换后的 start
            float[] genStart = new float[4];
            Matrix.multiplyMV(genStart, 0, invTransf, 0, r.start.toQici4(), 0);
            genRay.start.set(genStart);            //只取齐次坐标和前 3 个分量作为 start
            //求变换后的 dir
            float[] genDir = new float[4];
            Matrix.multiplyMV(genDir, 0, invTransf, 0, r.dir.toQici4(), 0);
            genRay.dir.set(genDir);                //只取齐次坐标和前 3 个分量作为 dir
        }
        //获取变换后的法向量(左乘变换矩阵的逆转置矩阵)
        public void xfrmNormal(Vector3 genNormal, float[] invTranspM, Vector3 normal){
            float[] tmpNormal = new float[4];        //用于存储变换后法向量的数组
            Matrix.multiplyMV(tmpNormal, 0, invTranspM, 0, normal.toQici4(), 0);
            genNormal.set(tmpNormal);               //只取齐次坐标和前 3 个分量作为 normal
        }
        public Point3 xfrmPtoPreP(Point3 P){ //通过变换后的点求变换前的点的方法: 乘以逆矩阵
            float[] inverM = getInvertMatrix();     //获取逆变换矩阵
            float[] preP = new float[4];            //存储变换前点坐标的数组
            Matrix.multiplyMV(preP, 0, inverM, 0, P.toQici4(), 0);   //求变换前的点
            return new Point3(preP);
        }
        public void initMyMatrix() {                //初始化变换矩阵的方法
            myMatrix = new float[16];
            Matrix.setIdentityM(myMatrix, 0);
        }
        public float[] getMatrix(){                 //获取对象的变换矩阵的方法
            return myMatrix;
        }
        public float[] getInvertMatrix(){           //获取逆矩阵的方法
            float[] invM = new float[16];           //用于存储结果矩阵的数组
            Matrix.invertM(invM, 0, myMatrix, 0);   //求逆矩阵
            return invM;
        }
        public float[] getInvertTransposeMatrix(){ //获取逆转置矩阵的方法
```

```java
51            float[] invTranspM = new float[16];              //用于存储结果矩阵的数组
52            Matrix.transposeM(invTranspM, 0, myMatrix, 0);  //求转置矩阵
53            Matrix.invertM(invTranspM, 0, invTranspM, 0);   //求逆转置矩阵
54            return invTranspM;
55        }
56        public void translate(float x, float y, float z) {   //设置沿x、y、z轴移动
57            Matrix.translateM(myMatrix, 0, x, y, z);
58        }
59        public void rotate(float angle, float x, float y, float z) { //设置指定轴旋转
60            Matrix.rotateM(myMatrix, 0, angle, x, y, z);
61        }
62        public void scale(float x, float y, float z) {       //设置沿x、y、z轴缩放
63            Matrix.scaleM(myMatrix, 0, x, y, z);
64        }
65    }
```

- 第4-6行为该类的成员变量，包括物体的颜色、物体的仿射变换矩阵及摄像机对象的引用。
- 第7-8行为该类的两个抽象方法，名称都为hit。第一个用于计算光线与物体的碰撞点，第二个用于计算光线与物体有没有交点（服务于阴影探测器），任何场景中的物体继承该类后都需要实现这两个方法。
- 第9-64行为场景中物体共同拥有的方法，任何继承了该类的物体都会拥有这些方法。合理利用继承可以减少大量重复的代码，同时提高代码的可维护性。

（13）接着就应该开发表示场景中物体的具体类了，本案例中主要有两个。一个是表示球体的Ball类，另一个是表示矩形平面的Square类。这里以球体类Ball为例进行介绍，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_9/com/bn/Sample15_9目录下的HitObject.java。

```
package com.bn.Sample15_9;                                    //声明包
public class Ball extends HitObject {                         //球心在原点，半径为 1 的标准球
    public Ball(Camera cam, Color3f color){                   //初始化颜色与摄像机
        this.cam=cam;this.color=color;
    }
    @Override
    public boolean hit(Ray r,Intersection inter) {  //重写的计算光线与物体碰撞点的方法
        Ray genRay=new Ray();                                    //变换后的光线
        xfrmRay(genRay, getInvertMatrix(), r);                   //计算出变换后的光线
            double A,B,C;                                        //计算交点的方程的 3 个系数
            A = Vector3.dot(genRay.dir,genRay.dir);              //求出第一个系数 A
            B = Vector3.dot(genRay.start, genRay.dir);  //求出第二个系数 B
            C = Vector3.dot(genRay.start, genRay.start)-1.0f;    //求出第三个系数 C
            double discrim = B*B-A*C;                             //求判别式的值
            if(discrim<0.0){ return false; }                     //若判别式值小于 0 则没有交点
            int num=0;                                           //目前的交点个数
            double discRoot = (float) Math.sqrt(discrim);    //求判别式的平方根
            double t1 = (-B-discRoot)/A;                         //第一个交点的对应相交时间
            if(t1>0.00001){
                inter.hit[0].hitTime=t1;                         //记录交点的时间
                inter.hit[0].hitObject=this;                     //记录交点所属的物体
                inter.hit[0].isEntering=true;
```

```
23                  inter.hit[0].surface=0;
24                  Point3 P = rayPos(r,t1);                //交点坐标(使用变换前的光线)
25                  inter.hit[0].hitPoint.set(P);           //变换后的交点位置
26                  Point3 preP = xfrmPtoPreP(P);           //通过变换后的点求变换前的点
27                  inter.hit[0].hitNormal.set(preP);       //变换前的点就是变换前的法向量
28                  num=1;                                  //交点数量
29              }
30              double t2 = (-B+discRoot)/A;                //第二个交点的对应相交时间
31              if(t2>0.00001){
32                  inter.hit[num].hitTime=t2;              //记录交点的时间
33                  inter.hit[num].hitObject=this;          //记录交点所属的物体
34                  inter.hit[num].isEntering=true;
35                  inter.hit[num].surface=0;
36                  Point3 P = rayPos(r,t2);                //交点坐标(使用变换前的光线)
37                  inter.hit[num].hitPoint.set(P);
38                  Point3 preP = xfrmPtoPreP(P);           //通过变换后的点求变换前的点
39                  inter.hit[num].hitNormal.set(preP);     //变换前的点就是变换前的法向量
40                  num++;                                  //交点数量
41              }
42              inter.numHits=num;
43              return (num>0);                 //若有一个或一个以上的有效交点则返回 true
44          }
45          @Override
46          public boolean hit(Ray r) {         //服务于阴影探测器判断是否在阴影中的方法
47              Ray genRay=new Ray();                               //变换后的光线
48              xfrmRay(genRay, getInvertMatrix(), r);              //获取变换后的光线
49              double A,B,C;                                       //计算交点的方程的 3 个系数
50              A = Vector3.dot(genRay.dir,genRay.dir);             //求出第一个系数 A
51              B = Vector3.dot(genRay.start, genRay.dir);          //求出第二个系数 B
52              C = Vector3.dot(genRay.start, genRay.start)-1.0f;   //求出第三个系数 C
53              double discrim = B*B-A*C;                           //求判别式值
54              if(discrim<0.0){ return false; }                    //若判别式小于 0 则没有交点
55              double discRoot = (float) Math.sqrt(discrim);       //求判别式的平方根
56              double t1 = (-B-discRoot)/A;   //第一次相交的时间
57              if(t1<0 || t1>1){               //若相交时间不在 0～1 内则不在阴影中
58                  return false;
59              }
60              return true;                    //否则在阴影中
61          }
62  }
```

- 第7-44行为实现的父类的用于计算光线跟物体交点的hit方法，其采用的算法在上一小节中已经进行了详细的介绍。请读者对照上一小节的原理阅读此方法的代码，这样便于理解。
- 第46-61行为服务于阴影探测器的计算交点的方法，其计算出相交时间后，判断时间值是否处于0～1的范围内，若在范围内，则对应点在阴影中，否则对应点不在阴影中。

提示

表示矩形平面的Square类的代码与上述Ball类的非常相似，这里不再赘述，需要的读者请参考随书光盘中的源代码。

（14）完成了物体类的开发后，下面就可以开发代表场景的Scene类了，其代码如下。

代码位置：见随书光盘中源代码/第15章/Sample15_9/com/bn/Sample15_9目录下的Scene.java。

```java
package com.bn.Sample15_9;                                   //声明包
……//此处省略了一些引入相关类的代码，需要的读者请参考随书光盘
public class Scene {
    Camera cam;                                               //摄像机
    Light light;                                              //光源
    Ray feeler = new Ray();                                   //阴影探测器
    List<HitObject> hitObjects;                               //物体列表
    Ball ball1;                                               //红色球
    Ball ball2;                                               //蓝色球
        Square sqare;                                         //矩形平面
        public Scene(Camera cam, Light light){
            this.cam=cam;                                     //初始化摄像机
            this.light = light;                               //初始化光源
            hitObjects = new ArrayList<HitObject>();          //初始化物体列表
            ball1 = new Ball(cam, new Color3f(BALL1_COLOR));//创建一个红色的球
            ball2 = new Ball(cam, new Color3f(BALL2_COLOR));//创建一个蓝色的球
            sqare = new Square(cam, new Color3f(PLANE_COLOR));//创建一个绿色的平面
            hitObjects.add(ball1);                            //将红色球加入物体列表
            hitObjects.add(ball2);                            //将蓝色球加入物体列表
            hitObjects.add(sqare);                            //将绿色矩形平面加入物体列表
        }
        public void transform(){                      //对场景中的物体进行变换，以摆放到位
            for(HitObject pObj:hitObjects){           //为所有物体初始化变换矩阵
                pObj.initMyMatrix();
            }
            sqare.rotate(-90, 1, 0, 0);               //旋转矩形平面
            sqare.scale(PLANE_WIDTH/2.0f, PLANE_HEIGHT/2.0f, 1);//缩放矩形平面
            ball1.translate(-CENTER_DIS, R, 0);           //平移红色球
            ball1.scale(R, R, R);                         //缩放红色球
            ball2.translate(CENTER_DIS, R, 0);            //平移蓝色球
            ball2.scale(R, R, R);                         //缩放蓝色球
        }
        public int shade(                                 //进行光线跟踪计算的方法
                Ray ray,                                  //光线
                Color3f color,                            //第一交点的颜色
                Point3 vetex,                             //第一交点的位置
                Vector3 normal                            //第一交点的法向量
        ){
            Intersection best = new Intersection();//用于保存到目前为止最近的交点
            getFirstHit(ray, best);                       //计算出第一交点
            if(best.numHits==0){                          //如果没有任何交点
                return -1;
            }
            color.set(best.hit[0].hitObject.getColor());       //交点处物体的颜色
            vetex.set(best.hit[0].hitPoint);                   //交点的位置
```

```
46            float[] inverTranspM = best.hit[0].hitObject.getInvertTransposeMatrix();
                                                                  //求逆转置矩阵
47            Vector3 preN = best.hit[0].hitNormal;               //变换前的法向量
48            //通过逆转置矩阵求变换后的法向量
49            best.hit[0].hitObject.xfrmNormal(normal, inverTranspM, preN);
50            Point3 hitPoint = best.hit[0].hitPoint;             //取出交点坐标
51            //阴影探测器的起点为：将交点朝人眼方向移动一个微小的距离
52            feeler.start.set(hitPoint.minus(ray.dir.multiConst(MNIMUM)));
53            feeler.dir = light.pos.minus(hitPoint);//阴影探测器的方向，从碰撞点指向光源
54            if(isInShadow(feeler)){                             //交点在阴影中返回 1
55                return 1;
56            }
57            return 0;                                           //交点不在阴影中返回 0
58        }
59        public void getFirstHit(Ray ray, Intersection best){//计算出此条光线与物体的最近交点
60            Intersection inter = new Intersection();            //创建交点列表对象
61            best.numHits=0;                                     //初始时交点数为 0
62            for(HitObject pObj:hitObjects){                     //遍历场景中的每一个物体
63                if(!pObj.hit(ray, inter)){                      //光线是否与当前物体相交
64                    continue;                                   //无交点则检测下一个物体
65                }
66                if(best.numHits==0 || //若 best 中还没有交点信息，或 best 中的交点不是最近点
67                        inter.hit[0].hitTime<best.hit[0].hitTime){
68                    best.set(inter);                            //将当前交点信息传入 best
69        }}}
70        //检测交点是否在阴影中的方法，入口参数为阴影探测器对应光线
71        public boolean isInShadow(Ray feeler){
72            for(HitObject pObj:hitObjects){             //遍历所有物体
73                if(pObj.hit(feeler)){                   //若光线被任何物体挡住，则在阴影中
74                    return true;
75            }}
76            return false;                               //否则不在阴影中
77    }}
```

- 第22-32行为对场景中物体进行变换的transform方法，其实现非常简单，就是调用各个物体对象的translate、rotate及scale方法对物体施加平移、旋转及缩放等仿射变换。
- 第33-58行为进行光线跟踪计算的shade方法，其主要功能为找到光线ray与场景中物体的第一个交点，并将交点的信息保存。最后返回一个整数表示计算结果的状态，具体含义为：-1表示没有交点；0表示有交点，且第一交点不在阴影中；1表示有交点，且第一交点在阴影中。
- 第59-69行为计算当前光线与场景中所有物体第一个交点的方法，其遍历场景中所有的物体单独计算，然后取时间值最小的交点。
- 第71-77行为检测交点是否在阴影中的方法，其参数为阴影探测器对应的光线。该方法的实现很简单，只要遍历所有物体的列表，调用每个物体对象的相应hit方法即可。

到这里为止，本节实现光线跟踪的案例就介绍完了。读者如果把基本块的大小设置为1，则可以发现光线跟踪是目前介绍的所有技术中光影效果最好的。但请读者注意的是，由于本书不是专门讨论光线跟踪的书籍，因此只是给出了一个非常简单的光线跟踪的实现。

另外，读者不要以为光线跟踪仅仅能实现光照与阴影效果，其一样可以实现反射、折射等光学效果，有兴趣的读者可以自行查阅相关资料完成。

15.10 本章小结

本章主要介绍了一些用于模拟现实世界中各种光学效果的技术，包括：投影贴图、反射、折射、色散、菲涅尔效果、凹凸映射、阴影、光线跟踪等。掌握了这些知识以后，读者若能恰当地灵活运用将可以开发出更具有吸引力与感染力的3D应用。

第16章

游戏开发中的物理学

本书前面的章节中介绍了很多构建与渲染呈现场景中物体的技术与技巧，掌握了这些知识后应该已经可以开发出很多酷炫的场景了。但仅靠这些想开发出具有吸引力的游戏应用还是远远不够的，还需要在程序中加入恰当的物理计算才能够真实地模拟现实世界。本章将向读者介绍一些游戏开发中常用的与物理学密切相关的知识，主要包括碰撞检测基本技术、穿透效应、粒子系统、弹簧质点系统等。

16.1　碰撞检测基本技术

游戏开发中经常遇到的一个问题就是物体与物体之间需要进行碰撞检测，例如英雄碰到障碍物就不能继续移动、静止的物体会被车撞飞等。进行碰撞检测的策略有很多，这里重点介绍的是最简单也是最常用的一种——AABB包围盒。

16.1.1　AABB包围盒的基本原理

进行碰撞检测时最直观和最精确的方式就是采用组成物体的三角形组来进行，将待检测的两个物体的三角形组中的三角形两两进行相交性检测，若有任意一对三角形相交则物体构成碰撞，否则物体不构成碰撞。这样做虽然思路很直观，但在一般设备上是难以实施的。

这是因为一般的物体都由大量的三角形构成，两两进行相交性检测需要非常大的计算量，很难满足游戏应用中实时性的需求。最直观的思路很难满足需求，此时可以采用的策略就是通过某种方式将问题简化，AABB包围盒（Axially Aligned Bounding Box简称AABB）就是一种很好的简化策略，其基本思路如图16-1所示。

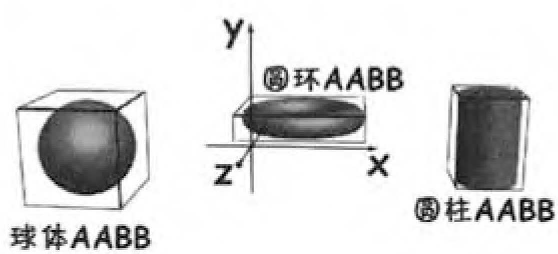


图16-1　不同物体的MBB

从图16-1中可以看出，所谓AABB包围盒就是采用一个长方体将物体包裹起来，进行两个物体的相交性检测时仅检测物体对应包围盒（包裹物体的长方体）的相交性。另外AABB包围盒有一个重要的特性，那就是包围盒对应的长方体的每一个面都是与某个坐标轴平面平行的，因此AABB包围盒又称为轴对齐包围盒。

有了上述约定后确定AABB包围盒就比确定任意姿态的长方体简单多了，仅需要记录6个值即可，这6个值分别代表包围盒在每个坐标轴上的最大值与最小值，即X_{min} 、X_{max} 、Y_{min} 、Y_{max} 、 Z_{min} 与Z_{max} 。也就是说，实际物体上所有的点都必须满足如下条件。

$$x_{min} \le x \le x_{max} \qquad y_{min} \le y \le y_{max} \qquad z_{min} \le z \le z_{max}$$

出于方便开发起见，可以将表示AABB包围盒的6个参数分为如下两组：

$$P_{min} = [x_{min}, y_{min}, z_{min}] \qquad P_{max} = [x_{max}, y_{max}, z_{max}]$$

> **说明**
>
> 其中P_{min}是3个轴坐标最小值的集合，P_{mzx}是3个轴坐标最大值的集合。

知道了表示AABB包围盒的6个参数后可以非常方便地求得AABB包围盒的几何中心c，公式如下。

$$c = (P_{min} + P_{max})/2$$

16.1.2 AABB包围盒的计算

了解了AABB包围盒的基本原理后，求AABB包围盒6个参数的方法也就跃然纸上了。仅需要对物体中的所有顶点坐标进行扫描，求出各个轴分量的最大值与最小值即可。为了方便开发，在介绍AABB包围盒的计算之前首先引入一个辅助类Vector3f，其对象可以用来记录3个浮点数组成的三元组，可以用来表示物体的位置、速度等，具体代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_1/com/bn/Sample16_1目录下的Vector3f.java。

```
package com.bn.Sample16_1;                                   //声明包
public class Vector3f{
    float x; float y; float z;                               //用于记录XYZ分量的成员变量
    public Vector3f(float x,float y,float z){
        this.x=x; this.y=y; this.z=z;                        //初始化XYZ分量
    }
    public void add(Vector3f temp){                          //增加XYZ分量值的方法
        this.x+=temp.x; this.y+=temp.y; this.z+=temp.z;
    }
}
```

提示

有了此辅助类后，物体的位置、速度等就可以直接用此类的对象表示了，这就比直接使用3个零散的变量要方便、易于维护。

了解了辅助类Vector3f后，接下来要介绍的是完成AABB包围盒相关计算的类AABBBox了，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_1/com/bn/Sample16_1目录下的AABBBox.java。

```
package com.bn.Sample16_1;                      //声明包
public class AABBBox{
    float minX; float maxX; float minY; float maxY; float minZ; float maxZ;
                                                //包围盒的6个参数
```

```
4     public AABBBox(float[] vertices){//通过传入的物体顶点序列计算出包围盒 6 个参数的构造器
5         init();                          //初始化包围盒的 6 个参数
6         findMinAndMax(vertices);    //根据物体的顶点序列计算出包围盒的 6 个参数值
7     }
8     //通过直接给出 6 个参数的值生成包围盒对象的构造器
9     public AABBBox(float minX,float maxX,float minY,float maxY,float minZ,float maxZ){
10            this.minX=minX; this.maxX=maxX; this.minY=minY;
11            this.maxY=maxY; this.minZ=minZ; this.maxZ=maxZ;
12        }
13        public void init(){                           //初始化包围盒的 6 个参数
14            minX=Float.POSITIVE_INFINITY; maxX=Float.NEGATIVE_INFINITY;
15            minY=Float.POSITIVE_INFINITY; maxY=Float.NEGATIVE_INFINITY;
16            minZ=Float.POSITIVE_INFINITY; maxZ=Float.NEGATIVE_INFINITY;
17        }
18        public void findMinAndMax(float[] vertices){//根据物体的顶点序列计算出包围盒的 6 个参数值
19            for(int i=0;i<vertices.length/3;i++){
                                  //对每个顶点进行循环求 X、Y、Z 3 个坐标的最大/小值
20                if(vertices[i*3]<minX){             //求 X 轴坐标的最小值
21                    minX=vertices[i*3];
22                }
23                if(vertices[i*3]>maxX){             //求 X 轴坐标的最大值
24                    maxX=vertices[i*3];
25                }
26                if(vertices[i*3+1]<minY){           //求 Y 轴坐标的最小值
27                    minY=vertices[i*3+1];
28                }
29                if(vertices[i*3+1]>maxY){           //求 Y 轴坐标的最大值
30                    maxY=vertices[i*3+1];
31                }
32                if(vertices[i*3+2]<minZ){           //求 Z 轴坐标的最小值
33                    minZ=vertices[i*3+2];
34                }
35                if(vertices[i*3+2]>maxZ){           //求 Z 轴坐标的最大值
36                    maxZ=vertices[i*3+2];
37        } } }
38        public AABBBox getCurrAABBBox(Vector3f currPosition){//获得物体平移后的 AABB 包围盒
39            AABBBox result=new AABBBox(         //根据变换后的位置重新生成 AABB 包围盒
40                this.minX+currPosition.x, this.maxX+currPosition.x, this.minY+curr
                  Position.y,
41                this.maxY+currPosition.y, this.minZ+currPosition.z, this.maxZ+curr
                  Position.z
42            );
43            return result;
44    } }
```

- 第13-17行为初始化AABB包围盒6个参数的init方法，其根据需要将最大值变量初始化为Float.NEGATIVE_INFINITY（负无穷大），将最小值变量初始化为Float.POSITIVE_INFINITY（正无穷大）。
- 第18-37行为根据传入的物体顶点坐标序列求出各个坐标轴分量最大值与最小值的findMinAndMax方法，其是计算包围盒6个参数的核心。
- 由于findMinAndMax方法求出的是物体原始位置情况下包围盒的6个参数，而应用程序运行时物体可能是需要移动的，因此需要有根据物体移动后的位置计算产生新包围盒对象的getCurrAABBBox方法

（第38-44行）。此方法入口参数的Vector3f对象中携带了物体移动后位置的3个坐标分量，用于与原始包围盒的6个参数进行运算产生移动后包围盒的6个参数。

> **提示**
>
> 将包围盒相关的操作及信息都封装到一个类中有利于代码的重用与维护，读者自己开发项目时也应该注意这一点。

16.1.3　AABB包围盒的碰撞检测

前面已经介绍过，求出物体AABB包围盒的目的是为了简化物体运动过程中的碰撞检测计算。因此，本小节将向读者介绍两个AABB包围盒碰撞检测的策略，其基本原理如图16-2所示。



图16-2　AABB包围盒碰撞检测的基本原理

从图16-2中可以看出，由于任何一个AABB包围盒的各个面都是平行于坐标平面的，因此判断两个AABB包围盒是否发生碰撞仅需要分别判断3个轴方向的交叠部分大小是否大于设定的阈值，若大于则发生了碰撞，否则没有发生碰撞。

> **说明**
>
> 合理设置阈值是非常重要的，阈值过小则可能给人的感觉还不应该碰撞时就发生了碰撞，而阈值过大就会产生两个物体视觉上已经交叠很多才检测出碰撞的情况。

了解了AABB包围盒碰撞检测的基本原理后，下面将介绍的是利用AABB包围盒进行碰撞检测的刚体类RigidBody，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_1/com/bn/Sample16_1目录下的RigidBody.java。

```
package com.bn.Sample16_1;                                      //声明包
import java.util.ArrayList;                                     //引入相关类
public class RigidBody{
    LoadedObjectVertexNormal renderObject;                      //加载的物体(渲染者)
    AABBBox collObject;                                         //AABB 包围盒(碰撞者)
    boolean isStatic;                                           //是否静止的标志位
    Vector3f currLocation;                                      //物体当前的位置
    Vector3f currV;                                             //物体当前的速度
    final float V_UNIT=0.5f;                                    //检测碰撞的阈值
        ……//此处省略了本类的构造器，读者可以自行查阅随书光盘中的源代码
        ……//此处省略了本类的绘制方法，读者可以自行查阅随书光盘中的源代码
        public void go(ArrayList<RigidBody> al){                //物体移动的方法
```

```
13          if(isStatic) return;                    //若为静止物体则不需要移动
14          currLocation.add(currV);                //物体根据速度值改变坐标
15          for(int i=0;i<al.size();i++){           //多列表中所有的刚体循环遍历
16              RigidBody rb=al.get(i);             //获得列表中的一个刚体
17              if(rb!=this){                       //若获得的刚体不是自己
18                if(check(this,rb)){               //检测两个刚体是否碰撞
19                  this.currV.x=-this.currV.x;     //若碰撞了，则 X 轴速度分量置反
20      }}}}
21      public boolean check(RigidBody ra,RigidBody rb){//检测两个刚体是否碰撞的方法
22        //计算两个刚体对应的 AABB 包围盒 3 个轴方向上交叠部分的大小
23        float[] over=calOverTotal(
24         ra.collObject.getCurrAABBBox(ra.currLocation),//获取 a 刚体当前位置对应的 AABB 包围盒
25         rb.collObject.getCurrAABBBox(rb.currLocation)//获取 b 刚体当前位置对应的 AABB 包围盒
26        );
27        //判断是否 3 个轴的交叠值都大于阈值
28        return over[0]>V_UNIT&&over[1]>V_UNIT&&over[2]>V_UNIT;
29      }
30      public float[] calOverTotal(AABBBox a,AABBBox b){//计算两个 AABB 包围盒 3 个轴的交叠值
31          float xOver=calOverOne(a.maxX,a.minX,b.maxX,b.minX);
32          float yOver=calOverOne(a.maxY,a.minY,b.maxY,b.minY);
33          float zOver=calOverOne(a.maxZ,a.minZ,b.maxZ,b.minZ);
34          return new float[]{xOver,yOver,zOver};
35      }
36      public float calOverOne(float amax,float amin,float bmax,float bmin){
                                          //求某个轴的交叠值
37          float minMax=0;               //两个最大值中较小的一个
38          float maxMin=0;               //两个最小值中较大的一个
39          if(amax<bmax){
40              minMax=amax;              //两个最大值中较小的为 amax
41              maxMin=bmin;              //两个最小值中较大的为 bmin
42          }
43          else {
44              minMax=bmax;              //两个最大值中较小的为 bmax
45              maxMin=amin;              //两个最小值中较大的为 amin
46          }
47          if(minMax>maxMin){            //若最大值中较小的大于最小值中较大的则有交叠
48            return minMax-maxMin;       //交叠值为最大值中较小的减去最小值中较大的
49          }else{
50              return 0;                 //否则没有交叠
51      }}}
```

- 第12-19行为物体根据当前速度移动一步的go方法，移动后要判断物体是否与其他刚体相撞，若相撞则将X轴速度置反。之所以一旦碰撞就将X轴速度置反是因为本节案例比较简单，物体是沿X轴方向运动的。若物体不仅是沿X轴方向运动则需要开发更为复杂的代码。
- 第21-29行为判断两个刚体是否碰撞的check方法，其首先将两个刚体当前位置对应的包围盒计算出来并送入calOverTotal方法计算两个AABB包围盒3个轴的交叠值，然后根据各个轴交叠值的情况判断是否发生了碰撞。

• 第36-51行为计算某个轴交叠值的calOverOne方法，其思路非常简单：首先求出两个AABB 包围盒此轴分量最大值中较小的与最小值中较大的，然后判断最大值中较小的是否大于最小值中较大的，若成立则将最大值中较小的减去最小值中较大的即可得到此轴的交叠值，否则没有发生交叠，交叠值为0。

上述刚体类RigidBody的成变量员中有两个最为重要的，渲染者renderObject与碰撞者collObjecto渲染者采用的是本书前面第9章介绍过的加载物体类，碰撞者是前面小节介绍的AABB包围盒类，其各自的作用如下。

• 渲染者。

渲染者主要负责将刚体渲染呈现出来，可以是任意复杂的形状。要注意的是其只负责渲染呈现的工作，不负责参与碰撞检测。

• 碰撞者。

碰撞者用于计算两个刚体是否发生了碰撞，其往往采用的是某种包围盒，如本节中采用的就是AABB包围盒。用于充当碰撞者的包围盒往往是从渲染者的顶点数据中计算产生的，其形状往往比渲染者简单很多，便于进行碰撞检测。

由渲染者负责渲染呈现，由碰撞者负责进行碰撞检测是3D游戏开发中一种非常常用的思路。采用这种思路既能很好地满足视觉效果的要求，又能很好地满足性能方面的要求。由渲染者和碰撞者共同组成刚体的情况如图16-3所示。

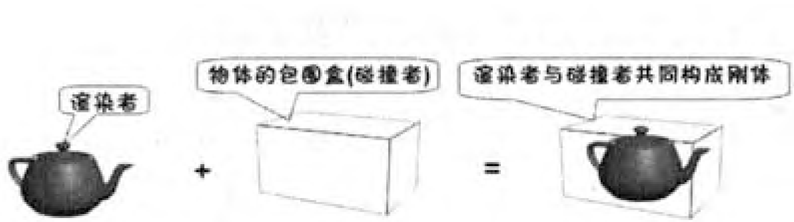


图16-3　渲染者、碰撞者以及组合而成的刚体

说明

图16-3中的渲染者为一只茶壶，若直接通过其进行碰撞检测计算量会非常大，而采用由渲染者导出的AABB包围盒进行碰撞检测计算量就小很多了，性能自然也就有了保障。

16.1.4　一个AABB包围盒的案例

前面几个小节介绍了AABB的基本原理与算法、Vector3f辅助类、AABB包围盒类以及刚体类。接下来就可以基于这些知识开发出一个在刚体运动中采用AABB包围盒进行碰撞检测的简单案例Sample16_1。在介绍开发步骤之前请读者先了解一下本案例的运行效果，如图16-4所示。

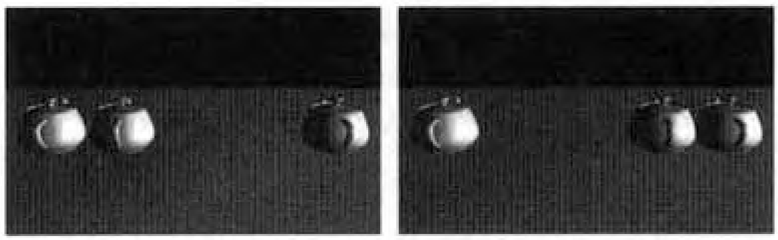


图16-4　案例Sample16_1运行效果图

> **说明**
>
> 从图16-4中可以看出，本案例中一共有3个刚体（3只茶壶）。其中左右两侧的是静止的，中间的在左右两侧的之间直线运动。当碰到某一侧的静止刚体时则改变方向返回，因此整个应用程序的运行过程中中间的茶壶是来回往复运动的。

了解了案例的运行效果后就可以进行案例的开发了，具体步骤如下。

（1）首先用3DMax生成两个物体（平面、茶壶），并导出成obj文件放入项目的assets目录中待用。

（2）开发出搭建场景的基本代码，包括：加载物体、摆放物体、计算光照等。这些代码与前面章节许多案例中的基本套路完全一致，因此这里不再赘述。

（3）开发出AABB包围盒类AABBBox以及刚体类RigidBody，这在前面的小节中已经介绍过。

（4）在MySurfaceView类中增加生成各个刚体并放进刚体列表的代码，具体内容如下。

代码位置：见随书光盘中源代码/第16章/Sample16_1/com/bn/Sample16_1目录下的MySurfaceView.java。

```java
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    ……//此处省略了一些与本书前面很多案例相同的代码，需要的读者请参考光盘
    //加载要绘制的物体，包括一个茶壶与一个平面
    ch=LoadUtil.loadFromFile("ch.obj", MySurfaceView.this.getResources(),
    MySurfaceView.this);
    pm=LoadUtil.loadFromFile("pm.obj", MySurfaceView.this.getResources(),
    MySurfaceView.this);
    aList.add(new RigidBody(ch,true,new Vector3f(-13f,0f,0),new Vector3f(0,0,0)));
                                            //生成左侧、右
    aList.add(new RigidBody(ch,true,new Vector3f(13f,0f,0),new Vector3f(0,0,0)));
                                            //侧及中间的刚体
    aList.add(new RigidBody(ch,false,new Vector3f(0f,0f,0),new Vector3f(0.1f,0,0)));
                                            //并放进刚体列表
    lgt=new LovoGoThread(aList);            //创建定时运动刚体的线程
        lgt.start();                        //启动定时运动刚体的线程
 }
```

> **说明**
>
> 注意3个刚体中前两个是静止的，只有最后一个是非静止的，因此其速度的X分量不为0。

（5）最后需要开发的是定时运动刚体的线程类LovoGoThread，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_1/com/bn/Sample16_1目录下的LovoGoThread.java。

```
package com.bn.Sample16_1;                                  //声明包名
import java.util.ArrayList;                                 //导入相关类
public class LovoGoThread extends Thread{
    ArrayList<RigidBody> al;                                //存储刚体列表
    boolean flag=true;                                      //线程控制标志位
    public LovoGoThread(ArrayList<RigidBody> al){
        this.al=al;                                         //线程中循环的标志位
    }
    public void run(){                                      //继承 Thread 类重写的 run 方法
        while(flag){                                        //while 循环
            int size=al.size();                             //获取列表中刚体的数量
            for(int i=0;i<size;i++){
                al.get(i).go(al);                           //调用 go 方法，使得刚体运动
            }
            try{sleep(10);}catch(Exception e){e.printStackTrace();}
    } } }
```

说明

从上述代码中可以看出，此线程的功能非常简单，其定时扫描刚体列表，调用每个刚体的go方法使得刚体按照当前的速度前进一步。

16.1.5　旋转后的AABB包围盒

前面的案例Sample16_1中仅考虑了平移对AABB包围盒的影响，但实际开发中物体在运动时不但有平移还有旋转。因此本节将讨论一下物体旋转后AABB包围盒的两种不同处理方案，具体内容如下。

1．直接遍历旋转后的顶点集合

此种方案的思路非常简单，每次物体旋转到新的姿态后，对旋转后物体的顶点序列重新进行扫描以生成新AABB包围盒的6个参数，如图16-5所示。

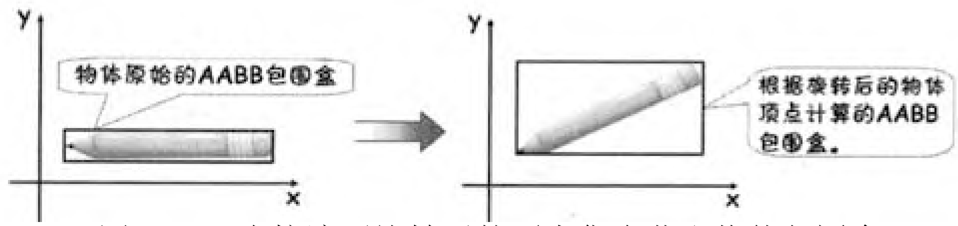


图16-5　直接遍历旋转后的顶点集合获取物体包围盒

提示

此种方案虽然简单，但在实际中较少使用。主要是因为需要将所有顶点旋转后的坐标都计算出来再进行遍历，计算量较大，难以满足程序实时性的要求。

2．基于旋转后的AABB包围盒求旋转后物体的AABB包围盒

此方案的思路也非常简单，首先根据物体旋转的情况将其原始AABB包围盒也做相同的旋转，然后基于旋转后AABB包围盒的顶点坐标再求旋转后物体的AABB包围盒，如图16-6所示。

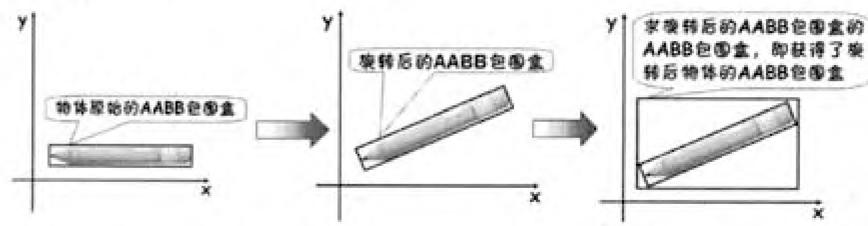


图16-6　旋转包围盒示意图

说明

从图16-6与图16-5的对比中可以发现，基于此方案求出的包围旋转后物体的AABB包围盒会不太紧密，有一些误差。但由于此方案的计算量相比于上一种方案小很多，误差也不是很大，因此实际开发中选用此方案的情况较多。

了解了如何求旋转后物体的AABB包围盒的两种方案后，下面基于第二种方案给出一个实现的案例Sample16_2，其运行效果如图16-7所示。

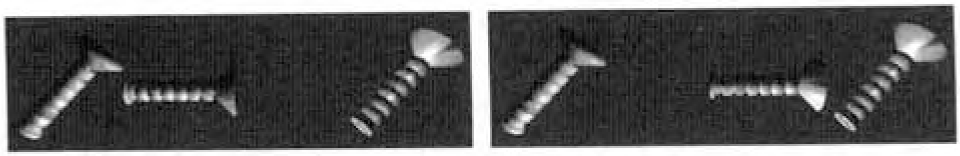


图16-7　旋转角度不为90°的运行效果图

从图16-7中可以看出，此案例中为了能说明旋转后AABB包围盒的问题，采用的加载物体不再是比较方正的茶壶，而是一个长条形的零件。同时从图中可以看出，零件也不都是按照坐标轴的方向摆放的，左右两边的零件都是旋转一定角度后摆放的。

若读者采用真机运行本案例就会发现，旋转后的AABB包围盒并不能将物体紧紧包裹（这一点从图16-6中也可以看出），因此物体碰撞检测的误差较大，会出现视觉上还没有碰撞程序就检测出碰撞的情况。如果物是旋转90°的整数倍摆放就要好很多了，如图16-8所示。

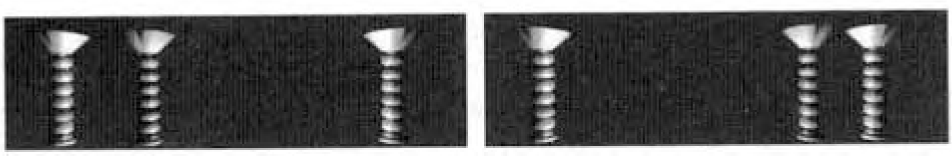


图16-8　旋转角度为90°的运行效果图

提示

若读者将物体摆放旋转角度改为90°再运行此案例，就会看到图16-8所示的情况，碰撞检测基本是比较准确的。

了解了本案例的运行效果后，接下来将介绍本案例的具体开发步骤。由于本案例只是将前面的案例Sample16_1复制并进行了升级，因此这里仅给出升级的主要步骤，具体如下所列。

（1）由于本案例相比于升级前增加了基于物体的旋转同时旋转AABB包围盒的功能，因此需要增加一个用于记录物体旋转姿态的辅助类Orientation，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_2/com/bn/Sample16_2目录下的Orientation.java。

```
package com.bn.Sample16_2;                                    //声明包名
import android.opengl.Matrix;                                  //引入相关类
public class Orientation{
    float[] orientationData=new float[16];         //用于记录物体旋转姿态的 4×4 矩阵
    //根据旋转角度及旋转轴向量的 XYZ 分量生成对应的 4×4 旋转矩阵
    public Orientation(float angle,float zx,float zy,float zz){
        Matrix.setRotateM(orientationData, 0, angle, zx, zy, zz);
} }
```

（2）由于需要将物体原始姿态下对应的AABB包围盒根据当前的姿态进行旋转求得旋转后物体的AABB包围盒，因此需要修改表示包围盒的AABBBox类，具体代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_2/com/bn/Sample16_2目录下的AABBBox.java。

```
package com.bn.Sample16_2;                    //声明包
import android.opengl.Matrix;                  //引入相关类
public class AABBBox{
    ......//此处省略了本类中成员变量的声明，读者可以自行查阅随书光盘中的源代码
    ......//此处省略了本类中通过直接给出 6 个参数的值生成包围盒对象的构造器
```

```
public AABBBox(float[] vertices){
    float[] data=findMinAndMax(vertices);        //找到每个轴的最大与最小值并返回
    minX=data[0];maxX=data[1]; //将每个轴的最大或最小值记录到对应的成员变量中
    minY=data[2];maxY=data[3]; minZ=data[4];maxZ=data[5];
    }
    public void init(float[] data){                //给出每个轴最大或最小值的初始值
        data[0]=Float.POSITIVE_INFINITY; data[1]=Float.NEGATIVE_INFINITY;
        data[2]=Float.POSITIVE_INFINITY; data[3]=Float.NEGATIVE_INFINITY;
        data[4]=Float.POSITIVE_INFINITY; data[5]=Float.NEGATIVE_INFINITY;
    }
    public float[] findMinAndMax(float[] vertices){
        float[] result=new float[6];               //创建记录每个轴最大或最小值的数组
        init(result);                              //给出每个轴最大或最小值的初始值
        for(int i=0;i<vertices.length/3;i++){
            if(vertices[i*3]<result[0]) {          //求 X 轴的最小值
                result[0]=vertices[i*3];
            }
            if(vertices[i*3]>result[1]) {          //求 X 轴的最大值
                result[1]=vertices[i*3];
            }
            if(vertices[i*3+1]<result[2]){         //求 Y 轴的最小值
                result[2]=vertices[i*3+1];
            }
            if(vertices[i*3+1]>result[3]){         //求 Y 轴的最大值
                result[3]=vertices[i*3+1];
            }
            if(vertices[i*3+2]<result[4]){         //求 Z 轴的最小值
                result[4]=vertices[i*3+2];
            }
            if(vertices[i*3+2]>result[5]){         //求 Z 轴的最大值
                result[5]=vertices[i*3+2];
        }}
        return result;
    }
    public AABBBox getCurrAABBBox(Vector3f currPosition,Orientation currOrientation){
        Vector3f[] va={                            //原始 AABB 包围盒 8 个顶点的坐标
            new Vector3f(minX,minY,minZ), new Vector3f(minX,maxY,minZ),
            new Vector3f(maxX,minY,minZ), new Vector3f(maxX,maxY,minZ),
            new Vector3f(minX,minY,maxZ), new Vector3f(minX,maxY,maxZ),
            new Vector3f(maxX,minY,maxZ), new Vector3f(maxX,maxY,maxZ),
        };
       float[] boxVertices=new float[24];          //存储旋转后包围盒 8 个顶点坐标的数组
       int count=0;                                //顶点计数器
       //遍历存储原始 AABB 包围盒 8 个顶点坐标的数组得到旋转后的 8 个顶点的坐标
       for(int i=0;i<va.length;i++){
            float[] result=new float[4];
            float[] dot=new float[]{va[i].x,va[i].y,va[i].z,1};
            Matrix.multiplyMV(result, 0, currOrientation.orientationData,0,dot,
             0);
            boxVertices[count++]=result[0];boxVertices[count++]=result[1];
            boxVertices[count++]=result[2];
       }
```

```
            //求包围盒旋转后各个轴坐标的最小/最大值，也就得到了旋转后物体的 AABB 包围盒
            float[] data=findMinAndMax(boxVertices);
            AABBBox result=new AABBBox(//根据物体平移后的位置重新生成新的 AABB 包围盒
                data[0]+currPosition.x, data[1]+currPosition.x,
                data[2]+currPosition.y, data[3]+currPosition.y,
                data[4]+currPosition.z, data[5]+currPosition.z
            );
            return result;//返回基于原始 AABB 包围盒求得的考虑了旋转与平移后的 AABB 包围盒
} }
```

提示

最大的改动位于求对应于当前物体位置与姿态的AABB包围盒的getCurrAABBBox方法中，将原来仅考虑物体的平移改变为同时考虑物体的旋转与平移。

（3）在刚体类RigidBody中增加用于描述物体旋转的Orientation类型的成员变量currOrientation，并在绘制物体与进行碰撞检测时增加与旋转相关的代码。这部分非常简单，需要的读者请参考随书光盘中的源代码。

（4）在工具类MatrixState中增加插入自定义变换矩阵的insertSelfMatrix方法，供绘制刚体时使用，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_2/com/bn/Sample16_2目录下的MatrixState.java。

```
public static void insertSelfMatrix(float[] matrixTemp){
    float[] result=new float[16];         //创建用于存放结果变换矩阵的数组
    //将传入的自定义变换矩阵与原变换矩阵相乘以得到最终的变换矩阵
    Matrix.multiplyMM(result, 0, currMatrix, 0,matrixTemp, 0);
    currMatrix=result;                    //返回最终的变换矩阵
}
```

（5）修改MySurfaceView类中生成各个刚体并放进刚体列表的代码，具体内容如下。

代码位置：见随书光盘中源代码/第16章/Sample16_2/com/bn/Sample16_2目录下的MySurfaceView.java。

```
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    ……//此处省略了一些与本书前面很多案例相同的代码，需要的读者请参考光盘
    //加载要绘制的物体，包括一个茶壶与一个平面
    ch=LoadUtil.loadFromFile("ch.obj", MySurfaceView.this.getResources(),MySurface
    View.this);
    pm=LoadUtil.loadFromFile("pm.obj",MySurfaceView.this.getResources(),MySurfaceView.
     this);
    aList.add(new RigidBody(ch,true,new Vector3f(-18f,0f,0), //生成左侧的刚体并放进刚体列表
       new Vector3f(0,0,0),new Orientation(45,0,1,0)));
    aList.add(new RigidBody(ch,true,new Vector3f(20f,0f,0),//生成右侧的刚体并放进刚体列表
       new Vector3f(0,0,0),new Orientation(45,0,1,0)));
      aList.add(new RigidBody(ch,false,new Vector3f(0f,0f,0),
                                       //生成中间的刚体并放进刚体列表
          new Vector3f(0.1f,0,0),new Orientation(0,0,1,0)));
      lgt=new LovoGoThread(aList);          //创建定时运动刚体的线程
      lgt.start();                          //启动定时运动刚体的线程
  }
```

说明

从上述代码中可以看出，在创建刚体时不但指定了每个刚体的初始速度与位置，还指定了每个刚体的旋转姿态。另外读者还可以自行修改代码，以不同的物体旋转角度运行本案例，细致进行观察运行的效果以加深对带旋转功能的AABB包围盒碰撞检测的理解。

16.1.6　AABB包围盒的使用要点

从前面几个小节的介绍中读者应该会发现一个规律，AABB包围盒对于本身横平竖直的物体在平行于坐标轴摆放的情况下，计算碰撞检测的误差很小，但对于不规则形状的物体或本身横平竖直的物体随意倾斜摆放时，计算碰撞检测的误差就会比较大，如图16-9所示。

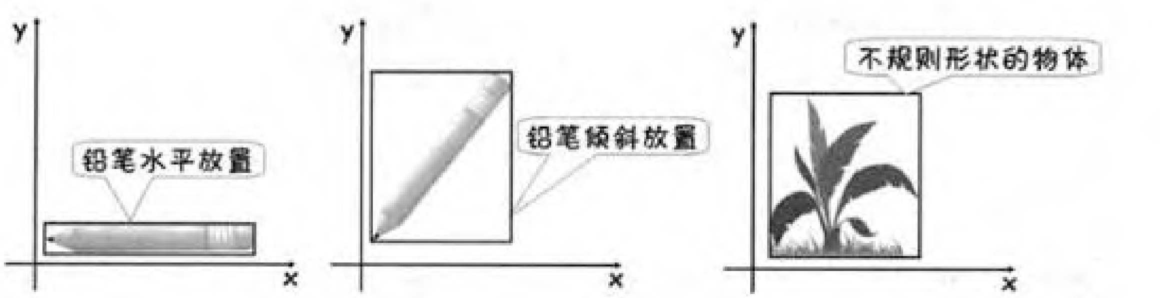


图16-9　两种姿态时的MBB包围盒

从图16-9中可以看出，对于同样物体不同姿态下的AABB包围盒以及不同形状的物体AABB包围盒，其物体本身区域占AABB包围盒区域的比例是大不相同的。物体本身区域所占比例越大，则碰撞检测的误差越小。因此在实际开发中应该尽量地提高物体本身区域所占的比例，以降低计算误差。

16.1.7 OBB包围盒

从前面几个小节对AABB包围盒的介绍中可以看出，物体旋转后AABB包围盒的计算误差可能会很大，如图16-9中倾斜的铅笔对应的AABB包围盒。如果希望改善这个问题，就可以使用OBB包围盒（Oriented Bounding Box简称OBB），其基本原理如图16-10所示。

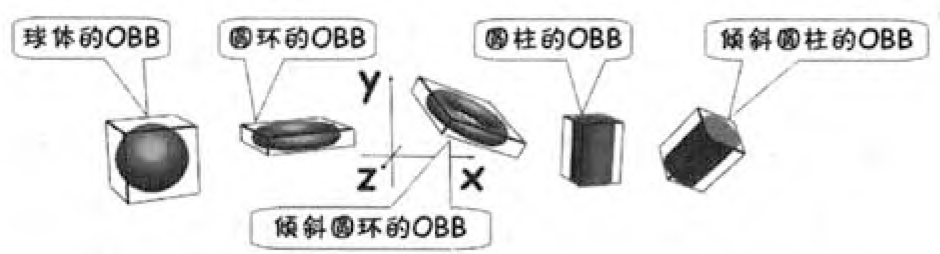


图16-10 几种物体的OBB包围盒

从图16-10中可以看出，OBB包围盒是随着物体的旋转而旋转的，并不需要总保持其6个面都平行于坐标平面，因此基于其进行的碰撞检测计算，大部分情况下误差远小于AABB包围盒。但检测两个OBB包围盒是否交叠的算法要比检测两个AABB包围盒是否交叠的算法复杂很多，因此在开发中最好是根据物体的形状及要求的碰撞检测精度来选用不同的包围盒，以达到性能与效果 兼顾的目的。

提示

由于基于OBB的碰撞检测算法比较复杂，大大超出了本书的讨论范围，因此本书中不再给出基于OBB进行碰撞检测的案例，有兴趣的读者可以参考其他相关资料自行开发。但请读者不要担心，实际开发中很少有完全有自己开发复杂碰撞的，一般都会使用物理引

擎来解决。下一章将向读者介绍常用的3D物理引擎之一——JBullet。

16.2 穿透效应

通过上一节的学习，读者应该基本掌握了AABB包围盒技术的使用。但由于前面一节给出的碰撞检测案例都是基于离散的位置进行计算的，若速度等参数设置不合理可能会产生穿透效应，本节将对这方面的内容进行介绍。

16.2.1 案例的运行效果与基本原理

现实世界中是不可能出现像神话故事“茅山道士”里的穿墙术的，但在虚拟的3D世界中若计算参数设置不合理，则可能会产生被称之为“穿透效应”的不合理效果。产生穿透效应的根本原因是因为，虽然应用程序运行时物体看起来是连续的，但实际上貌似连续的过程是由一系列离散的位置组成的。

因此有可能会产生这样的情况，两次离散位置之间有需要进行碰撞检测的物体，由于步进设置得比较大正好跨过了此物体，如图16-11所示。

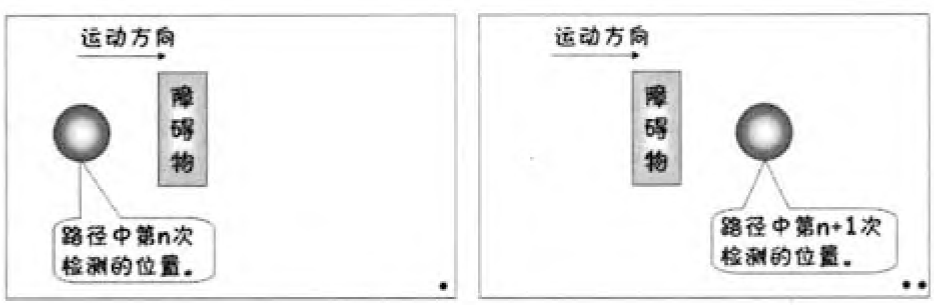


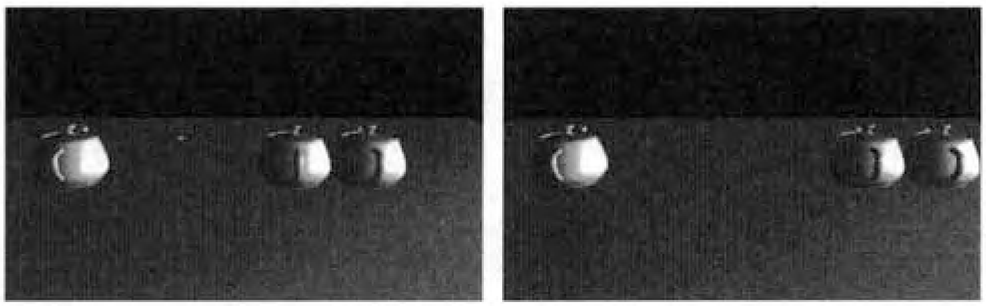
图16-11　穿透效应的成因

因此在开发中要特别注意与步进相关的一些参数的设置，如速度、每一步的时间等，只要设置合理就不会产生穿透效应，如图16-12所示。



图16-12　步进变小后不再有穿透效应了

了解了造成穿透效应的原因后，下面也给出一个由于参数设置不合理造成穿透效应的案例Sample16_3，其运行效果如图16-13所示。


图16-13　穿透效应案例的运行效果

说明

从图16-13中可以看出，此案例的场景与本章第一个案例相同，只不过由于步进设置比较大，中间运动的茶壶没有能检测出与右侧茶壶的碰撞，而是穿透过去了。

16.2.2　具体开发步骤

上一小节介绍了造成穿透效应的原因及其案例的运行效果，下面对本节案例的开发进行简要介绍。由于本节案例实际上仅仅是将案例Sample16_1复制了一份，并进行了简单的修改，因此这里仅给出需要修改的部分代码，具体内容如下。

代码位置：见随书光盘中源代码/第16章/Sample16_3/com/bn/Sample16_3目录下的MySurfaceView.java。

```
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    ……//此处省略了部分没有变化的代码，读者可以自行查阅随书光盘
    aList.add(new RigidBody(ch,true,
          new Vector3f(-14f,0f,0),new Vector3f(0,0,0)));  //生成左侧的刚体并放进刚体列表
    aList.add(new RigidBody(ch,true,
          new Vector3f(10f,0f,0),new Vector3f(0,0,0)));   //生成右侧的刚体并放进刚体列表
    aList.add(new RigidBody(ch,true,
          new Vector3f(-8f,0f,0), new Vector3f(12,0,0))); //生成中间运动的刚体并放进刚体列表
    lgt=new LovoGoThread(aList);                          //创建物体运动线程对象
       lgt.start();                                       //开启本线程
  }
```

提示

从上述代码中可以看出，主要是修改了中间运动刚体的速度，将原来的0.1修改为了12，其他的就是修改了几个刚体的初始位置。有兴趣的读者可以修改不同的速度来运行本案例，以加深理解。

16.3 粒子系统

现实世界中经常看到的焰火、喷泉等都可以看作是由许多粒子以不固定的速度向各个方向飞射组成的。本节中将要介绍的粒子系统就是基于这种思路开发的，可以用来模拟类似于焰火以及喷泉等效果。

16.3.1 案例运行效果与基本原理

介绍本节案例的具体开发步骤之前，首先需要了解一下本节案例的运行效果与基本原理，其运行效果如图16-14所示。

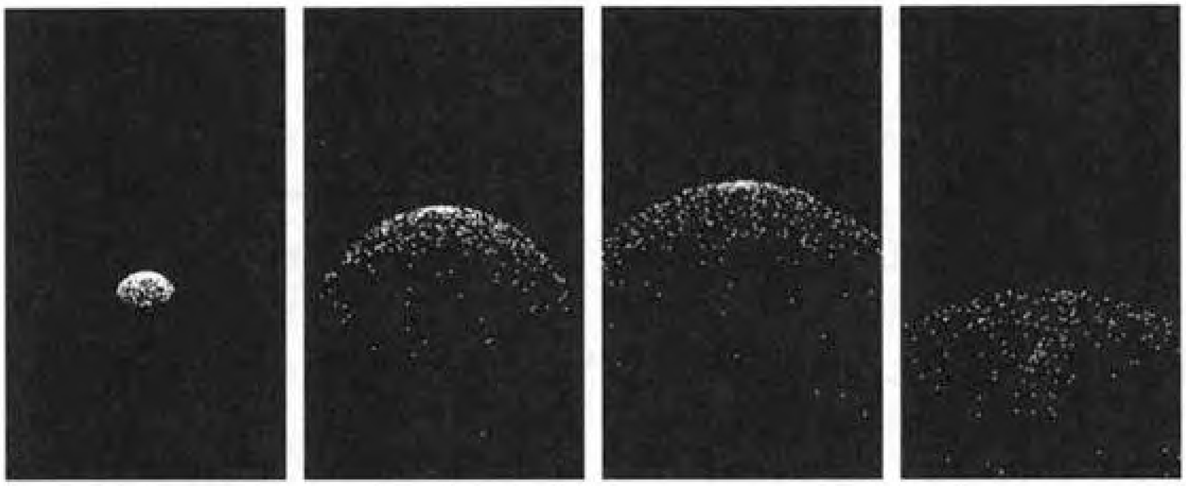


图16-14　案例Sample16_4运行效果

说明

图16-14中从左至右分别为一组粒子从起始点开始以不同的初速度发射出来然后逐步飞散过程中不同时刻的画面，建议读者用真机运行体会。

从图16-14中可以看出，一个粒子系统中可以同时存在多个粒子，这些粒子各自运动，互不影响。本案例中的粒子都是从相同的位置发射出来的，但初速度的大小与方向各不相同。发射时随机确定各个粒子的初速度方向与大小，如图16-15所示。

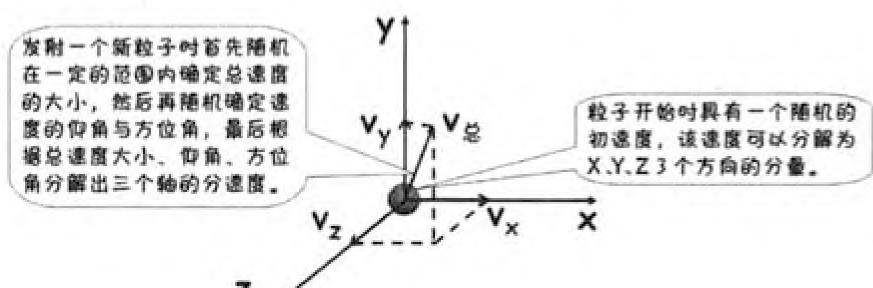


图16-15　粒子开始时的随机速度

粒子发射后需要按照牛顿运动定律继续运动才能达到模拟现实世界的效果，运动时为了计算方便，将X、Y、Z轴的位移分开独立计算。

假定重力加速度沿Y轴负方向，则具体情况如下。

- X轴、Z轴方向的运动不受重力加速度的影响，为最简单的匀速直线运动，运动的位移与时间及速度成正比。
- Y方向可以看作是上抛运动，位移需要采用如下公式进行计算。

$$S=Y_0+V_y\times t-0.5\times g\times t^2$$

本节案例中粒子发射后，各个轴就按照上述规则计算位移，具体过程如图16-16所示。

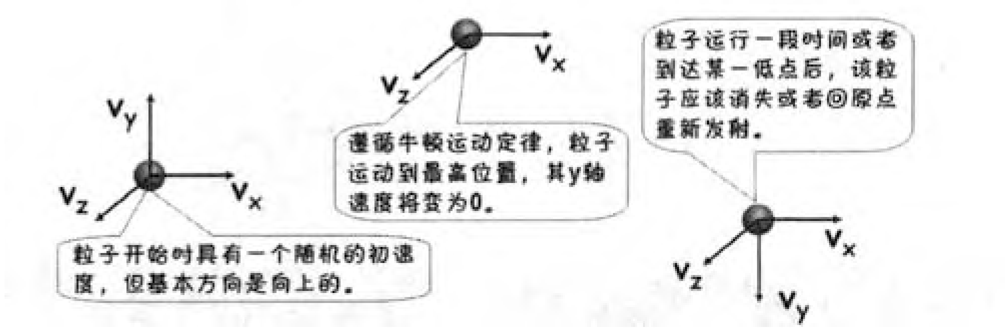


图16-16　粒子运动的基本过程

从图16-16中可以看出，粒子发射后遵循牛顿运动定律不断运动。但由于屏幕尺寸有限，当粒子运动一定的时间后，就会超出屏幕范围，此时就让粒子回原点重新开始一轮新的运动。

16.3.2　CPU版案例的开发

上一小节中介绍了本案例的运行效果与基本原理，下面将对采用CPU进行粒子物理计算的案例Sample16_4进行简要的介绍。由于本案例中的大部分类与本书前面章节很多案例中的非常类似，因此这里仅给出本案例中有代表性的部分，具体内容如下所列。

（1）首先给出的是表示单个粒子的SingleGrain类，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_4/com/bn/Sample16_4目录下的SingleGrain.java。

```
package com.bn.Sample16_4;                                   //声明包名
public class SingleGrain {
    float vx; float vy; float vz;                            //粒子在 x、y 以及 z 轴的速度分量
    float timeSpan=0;                                        //粒子的累计运动时间
    public SingleGrain(float vx,float vy,float vz){
        this.vx=vx; this.vy=vy; this.vz=vz;                  //初始化粒子在各个坐标轴上的速度
    }
    public void drawSelf(){
        MatrixState.pushMatrix();                            //保护现场
            float x=vx*timeSpan;                             //根据当前时间计算出粒子在
            float z=vz*timeSpan;                             //X 轴、Y 轴、Z 轴上的坐标
            float y=vy*timeSpan-0.5f*timeSpan*timeSpan*1.0f;
            MatrixState.translate(x, y, z);                  //平移坐标系
            GrainGroup.gfd.drawSelf();                       //绘制粒子
            MatrixState.popMatrix();                         //恢复现场
    }}
```

说明

上述代码中最主要的就是drawSelf方法，其首先根据牛顿运动定律计算出粒子的当前位置，然后将坐标系平移到需要的位置并绘制粒子。

（2）接着给出的是用来管理一组粒子的GrainGroup类，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_4/com/bn/Sample16_4目录下的GrainGroup.java。

```
package com.bn.Sample16_4;                                   //声明包名
import java.util.ArrayList;                                  //引入相关类
public class GrainGroup {                                    //代表粒子系统的类
    static GrainForDraw gfd;                                 //用于绘制粒子的绘制者
    ArrayList<SingleGrain> al=new ArrayList<SingleGrain>();           //所有粒子的列表
    static final float SPEED_SPAN= (float) (1.5f+1.5f*Math.random()); //粒子初速度
    static final float SPEED=0.02f;          //粒子移动每一步的模拟时延，也就是时间间隔
```

```
8    public GrainGroup(MySurfaceView mv){
9        gfd=new GrainForDraw(2,1,1,1,mv);            //创建粒子的绘制者
10           for(int i=0;i<400;i++){                  //随机向列表中添加不同初速度的粒子
11               double elevation=0.35f*Math.random()*Math.PI+Math.PI*0.15f;  //仰角
12               double direction=Math.random()*Math.PI*2;          //方位角
13               float vy=(float)(SPEED_SPAN*Math.sin(elevation));//分解出 3 个轴的初速度
14               float vx=(float)(SPEED_SPAN*Math.cos(elevation)*Math.cos(direction));
15               float vz=(float)(SPEED_SPAN*Math.cos(elevation)*Math.sin(direction));
16               al.add(new SingleGrain(vx,vy,vz)); //创建粒子对像并添加进粒子列表
17       } }
18       long timeStamp=0;                                  //用于计算的时间戳
19       public void drawSelf(){
20           long currTimeStamp=System.nanoTime()/1000000;   //获取当前系统时间
21           if(currTimeStamp-timeStamp>10){     //若时间间隔大于 10ms，则各个粒子前进一步
22               for(SingleGrain sp:al){          //扫描粒子列表，并修改粒子的累计运动时间
23                   sp.timeSpan=sp.timeSpan+SPEED;
24                   if(sp.timeSpan>10){          //判断粒子的累计运动时间是否大于 10
25                       sp.timeSpan=0;           //将粒子的累计运动时间归零
26               } }
27               timeStamp=currTimeStamp;         //更新用于计算的时间戳
28           }
29           int size=al.size();
30           for(int i=0;i<size;i++){                //循环扫描所有粒子的列表并绘制各个粒子
31            try{ al.get(i).drawSelf();}catch(Exception e){} //绘制粒子
32   } } }
```

- 第10-17行通过循环随机生成了400个初速度不同的粒子并加入到了列表中供后面使用。
- 第19-32行的drawSelf方法中每次绘制粒子组前都根据时间戳的情况更新粒子的累计运动时间，若粒子的累计运动时间超过一定的阈值则将其归0，最后一一绘制粒子列表中的所有粒子。

> **提示**
>
> 超过阈值将粒子的累计运动时间归0相当于重新从原点处发射了一个新粒子，这样程序运行中每隔一段时间都会有一批新的粒子从原点以不同的初速度发射出来，读者可以运行本案例自行体会。

16.3.3　GPU版案例开发步骤

上一小节给出的粒子系统案例中各个粒子的物理计算是采用CPU完成的，随着粒子数目的增大程序的运行速度会急剧下降，也就是

FPS（帧速率）快速降低。这是由CPU本身的特性所决定的，CPU采用是顺序执行的硬件架构，自然对粒子数量非常“敏感”。

为了大大提高程序的运行速度，本小节给出一个粒子系统的GPU版本Sample16_5，此案例中的物理计算都是采用顶点着色器完成的。由于本案例中很多代码与上一小节的案例非常类似，因此这里仅给出具有代表性的部分，具体内容如下所列。

（1）首先给出的是完成单个粒子物理计算的顶点着色器，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_4/assets目录下的vertex_yh.sh。

```
uniform mat4 uMVPMatrix;                        //总变换矩阵
uniform float uPointSize;                       //点尺寸
uniform float uTime;                            //粒子的累计运动时间
attribute vec3 aVelocity;                       //粒子初速度
void main(){
  float currTime=mod(uTime,10.0);               //执行取模运算，相当于累计时间超过10则归0
  float px=aVelocity.x*currTime;                //计算粒子此时的X坐标
  float py=aVelocity.y*currTime-0.5*1.0*currTime*currTime+3.0; //计算粒子此时的Y坐标
  float pz=aVelocity.z*currTime;                //计算粒子此时的Z坐标
     gl_Position = uMVPMatrix * vec4(px,py,pz,1);//根据总变换矩阵计算此次绘制此顶点的位置
     gl_PointSize=uPointSize;                   //设置粒子的尺寸
  }
```

说明

该顶点着色器的主要功能为根据渲染管线传入的粒子初速度及累计运动时间计算出粒子当前的位置并传递给管线。

（2）接着给出的是代表一组粒子的GrainForDraw类，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_5/com/bn/Sample16_5目录下的GrainForDraw.java。

```
1package com.bn.Sample16_5;                          //声明包名
2......//此处省略了部分引入相关类的代码，读者可以自行查阅随书光盘中的源代码
3public class GrainForDraw{                           //用户绘制粒子系统的类
4     private FloatBuffer   mVelocityBuffer;         //顶点速度数据缓冲
5     ......//此处省略了本案例中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
6    int maVelocityHandle;                            //顶点速度属性引用
7    float timeLive=0;                                //累计运动时间
8    long timeStamp=0;                                //用于计算的时间戳
9    ......//此处省略了本类的构造器，读者可以自行查阅随书光盘中的源代码
10        public void initVertexData(int vCount){   //初始化顶点数据的方法
11            float[] velocity=new float[vCount*3];//顶点速度数据数组，每个顶点 3 个速度分量
12            for(int i=0;i<vCount;i++){              //循环生成各个顶点的速度数据
13                   double fwj=2*Math.PI*Math.random();//随机生成方位角
14                   double yj=0.35*Math.PI*Math.random()+0.15*Math.PI;//随机生成仰角
15                   final double vTotal=1.5+1.5*Math.random();       //随机生成总速度
16                   double vy=vTotal*Math.sin(yj);                    //y 方向上的速度
17                   double vx=vTotal*Math.cos(yj)*Math.sin(fwj);     //x 方向上的速度
18                   double vz=vTotal*Math.cos(yj)*Math.cos(fwj);     //z 方向上的速度
19                   velocity[i*3]=(float)vx; velocity[i*3+1]=(float)vy; velocity
                     [i*3+2]=(float)vz;
20            }
26            ......//此处省略了将顶点初速度数据添加到缓冲区的代码，读者可以自行查阅随书光盘
27        }
28        ......//此处省略了初始化着色器的 initShader 方法，读者可以自行查阅随书光盘中的源代码
29        ......//此处省略了本案例中自定义的绘制方法，读者可以自行查阅随书光盘中的源代码
30    }
```

说明

上述代码中最重要的就是第10-27行生成各个顶点（粒子）初速度数据的initVertexData方法，其采用的计算思路与上一小节中的案例完全相同。

到这里为止，CPU版与GPU版粒子系统的案例就都介绍完了，下面对两个不同版本的案例进行一下性能比较，具体情况如表16-1所列。

表16-1　CPU版粒子系统与GPU版粒子系统帧速率

粒子数量	100 个	200 个	400 个	800 个	1600 个	3200 个	100000 个
CPU 版帧速率	45	44	30	16	6	3	0.12
GPU 版帧速率	45	45	31	22	22	22	21

> 提示
>
> 笔者采用的是华为U8800运行测得的数据，若读者采用其他机型数据可能会有所不同，但总体趋势不变。

从表16-1中可以非常清楚地对比出，随着粒子数量的增加，CPU版本案例的FPS很快下降到了不能流畅运行的情况。而GPU版本的案例在粒子数量增加后虽然FPS也有所降低，但基本能够保证程序的正常运行。

之所以CPU与GPU版本会有如此大的性能差异就在于GPU是流式处理器，其中的各个顶点处理单元是独立计算的，同一时刻可以处理多个顶点（粒子），可以很好地支持高并发，这正符合本节案例中各个粒子独立运动的情况。而CPU一次只能处理一个顶点，自然对顶点数量的增加十分敏感。

> 提示
>
> 开发中将能够由GPU处理的问题交给GPU处理是一个常用的提高程序执行性能的技术手段，读者应该多总结、多应用。

16.4　弹簧质点模型

前面几节介绍的物理计算主要是基于刚体的，都不需要考虑形变的问题。但现实世界中还有很多软体，运动过程中会发生形变。本节将基于弹簧质点模型介绍软体的仿真，同时还会给出一个采用此模型模拟绳子运动的案例Sample16_6。

16.4.1　案例运行效果与基本原理

介绍本节案例的具体开发步骤之前，首先需要了解一下本案例的运行效果与基本原理，其运行效果如图16-17所示。



图16-17　案例Sample16_6运行效果

> **说明**
>
> 图16-17中从左至右为绳子一头被不断旋转提升的过程，由于插图是静态的，所以感觉可能不是很明显，请读者用真机运行本案例进行体会。

了解了案例的运行效果后，下面简单介绍一下本案例中模拟绳子运动的原理。前面已经提过，本节采用的是弹簧质点模型，其原理如图16-18所示。



图16-18　用于模拟绳子的弹簧质点模型

从图16-18中可以看出，弹簧质点模型中将绳子看作由弹簧连接起来的多个质点组成。模拟绳子运动的过程中通过对每一个质点进行经典力学的受力分析来得到组成绳子的每个质点的运动情况，具体内容如下所列（如图16-19所示）。

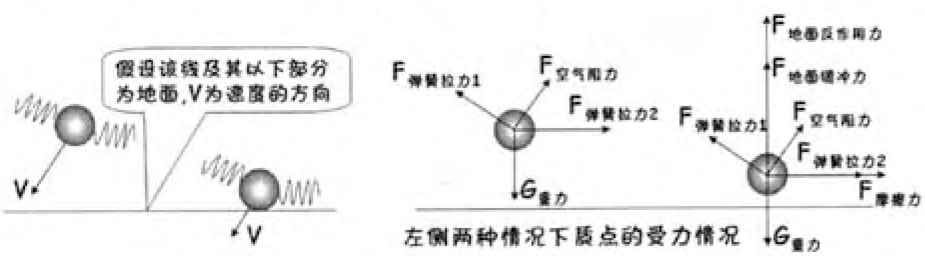


图16-19　运动质点的受力分析

- 当质点未与地面接触时，质点所受到的力包括左右弹簧对其的拉力、运动过程中空气的阻力以及自身的重力。

- 当质点与地面接触时，质点所受到的力包括左右弹簧对其的拉力、运动过程中空气的阻力、自身的重力，地面的摩擦力、地面的缓冲力以及地面的反作用力。

> **提示**
>
> 图16-19的两幅小图中，各自左侧的为在空中与地面无接触的质点，右侧的为与地面有接触的质点。另外需要注意的是，当质点处于绳子最左端或最右端时，质点所受到的弹簧拉力将减少一个。

了解了质点的受力情况后，下面来简单说明一下弹簧在运动过程中的受力情况。为了不让弹簧一直地振动下去，本模型的计算中还为弹簧指定了一个阻尼系数，使弹簧在运动的过程中还受到自身的阻尼力。各种不同力的计算方法如下所列。

- 弹簧阻尼力：$F=-u\cdot(v_1-V_2)$，其中v1和v2分别是弹簧两端质点的速度，u为阻尼系数。
- 弹簧拉力：F=-kx，其中k为弹簧的弹力系数，x是弹簧偏离平衡位置的位移。
- 空气阻力：F=-fv，其中f为空气阻力系数，v为质点的运动速度。
- 重力：G=mg，其中m为质点的质量，g为重力加速度。
- 地面摩擦力：$F=-uv_{平行}$，其中u为地面的摩擦系数，$v_{平行}$ 为质点速度在与地面平行方向上的分量。
- 地面的缓冲力：$F=-uv_{垂直}$，其中u为地面的缓冲系数，$v_{垂直}$直为质点速度在与地面垂直方向上 的分量。
- 地面的反作用力：$F=r(h_g-h_m)$，其中r为反作用力系数，h_g 为地面高度，h_m 为质点高度。

模拟的过程中，实时根据上述的系列公式计算出每个质点的位置，然后在绘制时每两个质点间用一根圆柱相连，即可渲染出绳子运动的过程，如图16-20所示。

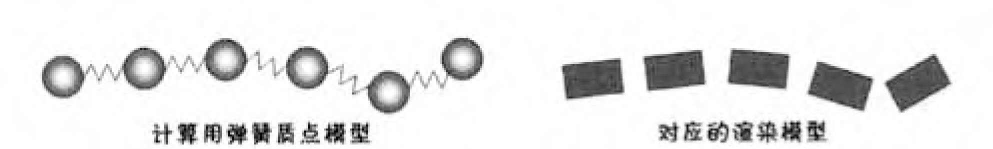


图16-20　渲染绳子的原理

> **提示**
>
> 本节案例的运行过程中通过一个线程不断地给绳子一端的质点施加变化的作用力，绳子就在这个力的牵引下运动起来了。

16.4.2　具体开发步骤

上一小节介绍了用于模拟绳子运动的弹簧质点模型及各种相关力的计算公式，本小节将简要介绍一下本节案例Sample16_6的开发步骤。由于本案例中与渲染相关的部分代码与本书前面很多案例中的非常相似，因此这里仅给出本案例中有代表性的与物理计算相关的部分，具体内容如下。

（1）首先给出的是用于描述质点的Mass类，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_6/com/bn/Sample16_6目录下的Mass.java。

```
1 package com.bn.Sample16_6;                          //包声明
2 public class Mass {
3     float m;                                        //质点的质量
4     Vector3 pos;                                    //质点的位置
5     Vector3 vel;                                    //质点的速度
6     Vector3 force;                                  //质点的受力
7     public Mass(float m){                           //构造器 1
8         this.m=m;                                   //初始化质量
9         pos = new Vector3();                        //初始化位置
10            vel = new Vector3();                    //初始化速度
11        }
12        public Mass(float m,Vector3 pos,Vector3 vel){        //构造器 2
13            this.m=m;                                        //初始化质量
14            this.pos=pos;                                    //初始化位置
15            this.vel=vel;                                    //初始化速度
16        }
17        public void applyForce(Vector3 force){  //对质点施加力的方法（包括重力、空气阻力等）
18            this.force=this.force.add(force);
19        }
20        public void initForce(){          //该方法初始化质点受到的力，初始时受到的力为 0
21            force = new Vector3(0,0,0);
22        }
23        public void calculateCurrPosAndVel(float dt){   //计算质点当前位置与速度的方法
24            Vector3 a = force.multiConstant(1/m);       //计算加速度 a = F/m
25            Vector3 deltaV = a.multiConstant(dt);       //计算速度增量 deltaV = a•dt
26            vel = vel.add(deltaV);                      //计算新速度 v = v + a•t
27            pos = pos.add(vel.multiConstant(dt));       //计算新位置 pos = pos + v•t
28    }}
```

- 第17-19行为给质点施加作用力的applyForce方法，其将新作用力与质点原来受的作用力进行向量合成以得到质点最新的总受力情况。
- 第20-22行为初始化质点受力的方法，初始时质点的总受力为零。
- 第23-28行是计算当前质点位置与速度的方法，该方法首先计算出质点的加速度，然后根据加速度计算出质点的速度增量，接着计算出质点的最新速度，然后根据质点的最新速度计算出质点的位移，最终得到质点的最新位置。

提示

上述代码中用到的力和速度均为矢量，用自定义的Vector3类的对象来表示。该类中封装了一些关于向量的基本操作，与前面章

节中的向量相关类非常类似。因此这里不再赘述，需要的读者可以参考随书光盘中的源代码。

（2）接着给出的是用于描述弹簧的Spring类，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_6/com/bn/Sample16_6目录下的Spring.java。

```
1 package com.bn.Sample16_6;                          //包声明
2 public class Spring {
3     Mass mass1;                                      //质点1(弹簧的一端)
4     Mass mass2;                                      //质点2(弹簧的另一端)
5     float springConstant;                            //弹力系数
6     float springLength;                              //弹簧长度
7     float frictionConstant;                          //阻尼系数
8     public Spring(Mass mass1, Mass mass2) {          //构造器
9         this.mass1 = mass1;                                          //初始化第一个质点
10            this.mass2 = mass2;                                      //初始化第二个质点
11            this.springConstant = Constant.springConstant;           //初始化弹力系数
12            this.springLength = Constant.springLength;               //初始化弹簧长度
13            this.frictionConstant = Constant.frictionConstant;       //指定弹簧阻尼系数
14        }
15        public void calculateSpringForce(){                   //计算弹簧及两端质点受力的方法
11            //计算弹簧的形变方向向量，由两端的质点位置确定
16            Vector3 springVector = mass1.pos.add(mass2.pos.multiConstant(-1));
17            float distance = springVector.length();     //此弹簧两端质点间的距离
18            Vector3 force;                              //作用力
19            if(distance!=0){
20                float deltaX = distance-this.springLength;  //弹簧偏离平衡位置的距离
21                Vector3 normalV =springVector.
22                 multiConstant(1/distance).multiConstant(-1);//将弹簧的形变方向向量规格化
23                force = normalV.multiConstant(deltaX).
24                 multiConstant(springConstant);     //计算出弹簧的拉力
25                force = force.add(mass1.vel.add(mass2.vel.multiConstant(-1)).
26                 multiConstant(-frictionConstant)); //计算出弹簧的阻尼力并与弹簧拉力形成合力
27                mass1.applyForce(force);        //将弹簧的拉力应用于一端的质点
28                mass2.applyForce(force.multiConstant(-1));//将弹簧的拉力应用于另一端的质点
29    }}}
```

- 第8-14行为此类的构造器，其用于对各个成员变量进行初始化，主要包括两端的质点、弹力系数、初始弹簧长度、阻尼系数等。
- 第15-31为计算弹簧拉力与阻尼力的方法，其首先根据弹簧的形变计算出拉力，然后根据两端质点的速度计算出阻尼力，最后将这两项作用力合成并分别施加到两端的质点上。需要特别注意的是，弹簧对两端质点的作用力是相反方向的。

（3）下面给出的是用于模拟绳子运动的抽象类RopeSimulation，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_6/com/bn/Sample16_6目录下的RopeSimulation.java。

```
package com.bn.Sample16_6;                              //包声明
……//此处省略了一些导入相关类的代码，读者可以自行查阅随书光盘中的源代码
public abstract class RopeSimulation {
    List<Mass> massList;                                //存放绳子中所有质点对象的列表
    int massCount;                                      //绳子中的质点数量
    public RopeSimulation(int massCount,float m){    //参数为质点的数量与每个质点的质量
        this.massCount=massCount;                       //初始化质点数量
        massList = new ArrayList<Mass>();               //创建存放所有质点对象的列表
        for(int i=0;i<massCount;i++){
                Mass mass = new Mass(m);                //创建 massCount 个质量均为 m 的质点
                massList.add(mass);                     //将质点加入到列表中
        }}
        public Mass getMass(int index){                 //获取质点对象列表中指定编号的质点对象
            if(index<0||index>=massCount){              //若不存在该质点
                return null;                            //返回 null
            }
            return massList.get(index);                 //返回相应的质点
        }
        public void init(){                             //初始化质点列表中每一个质点对象的受力情况
            for(int i=0;i<massList.size();i++){
                massList.get(i).initForce();            //初始化时每个质点的受力均为 0
        }}
        public abstract void solve();                   //计算绳子各质点和弹簧当前受力的方法
        public void simulateRope(float dt){             //更新质点列表中每一个质点当前的位置与速度
            for(int i=0;i<massList.size();i++){//遍历质点列表，对每个质点进行更新
                massList.get(i).calculateCurrPosAndVel(dt); //更新该质点的速度和位置
        }}
        public void operate(float dt){ //对整个绳子进行更新的方法
            init();                     //初始化
            solve();                    //计算绳子运动轨迹
            simulateRope(dt);           //计算每个质点的新位置与新速度
}}
```

说明

该类为模拟绳子运动的一个抽象类，真正需要模拟绳子时需要继承此类并实现其中的solve方法，实现类将在后面给出。

（4）下面要给出的是集成自RopeSimulation以实现模拟绳子运动的Rope类，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_6/com/bn/Sample16_6目录下的Rope.java。

```
package com.bn.Sample16_6;                                    //包声明
......//此处省略了一些导入相关类的代码，读者可以自行查阅随书光盘中的源代码
public class Rope extends RopeSimulation {
    List<Spring> springList;                                   //弹簧列表
    ......//此处省略部分成员变量声明的代码，读者可自行查阅随书光盘的源代码
    float airFrictionConstant;                                 //空气的摩擦系数
    public Rope(int massCount,float m,Vector3 gravity,float groundRepulsionConstant,
            float groundFrictionConstant,float groundAbsorptionConstant,float
            groundHeight,
            float airFrictionConstant,Vector3 connectionVel,Vector3 connectionPos) {
            super(massCount, m);       //调用父类的构造器，创建 massCount 个质量为 m 的质点
            this.gravity = gravity;                            //初始化重力加速度
            ......//此处省略了一些初始化成员变量的代码，读者可自行查阅随书光盘的源代码
            this.connectionPos=connectionPos;                  //绳头位置
            for(int i=0;i<massCount;i++){                      //初始化所有质点的位置
                massList.get(i).pos.x = i*springLength;//给出 x 坐标
                massList.get(i).pos.y = groundHeight;  //给出 y 坐标
                massList.get(i).pos.z = 0;             //给出 z 坐标
            }
            springList = new ArrayList<Spring>();              //创建存放弹簧对象的列表
            for(int i=0;i<massCount-1;i++){                    //初始化质点之间的弹簧
                Spring temp = new Spring(massList.get(i),massList.get(i+1));
                springList.add(temp);                          //加入到弹簧对象的列表
        }}
        public void solve() {                                  //计算绳子当前运动轨迹的方法
            for(int i=0;i<massCount-1;i++){//遍历弹簧列表，计算每个弹簧对其两端质点施加的力
                springList.get(i).calculateSpringForce();
            }
            for(int i=0;i<massCount;i++){  //遍历质点列表，计算质点受到的其他力
                Mass mass = massList.get(i);   //从列表中取出一个质点
                mass.applyForce(gravity.multiConstant(mass.m)); //计算并施加重力
                //计算并施加空气阻力
                mass.applyForce(mass.vel.multiConstant(-1).multiConstant
                (airFrictionConstant));
                if(mass.pos.y<=groundHeight){                  //质点当前的高度小于地面的高度时
                    Vector3 v = mass.vel.copy();               //复制当前质点的速度
                    v.y=0;                                     //y 方向的速度设为 0
                    //计算地面摩擦力并施加于质点
                    mass.applyForce(v.multiConstant(-1).multiConstant(ground
                    FrictionConstant));
                    v = mass.vel.copy();                       //复制当前质点的速度
                    v.x=0;   v.z=0;                            //x 与 z 方向的速度设为 0
                    if(v.y<0){                                 //若当前质点 y 方向的速度小于 0
                     mass.applyForce(v.multiConstant(-1).//计算地面缓冲力并施加于质点
                        multiConstant(groundAbsorptionConstant));
                    }                                          //给定地面的缓冲力
```

```java
44                  Vector3 force = new Vector3(0,groundRepulsionConstant,0).
45                   multiConstant(groundHeight-mass.pos.y);//计算地面的反作用力
46                  mass.applyForce(force);       //将地面反作用力施加于质点
47          }}}
48      public void simulateRope(float dt){          //模拟绳子运动的方法
49          super.simulateRope(dt);                  //调用父类的模拟方法
50          Mass head = massList.get(0);             //获取绳头的质点
51          Vector3[] result = calCentripetalForceAndVel();
                                                     //获取施加与绳头的作用力及绳头速度
52          head.applyForce(result[0]);              //施加向心力
53          connectionVel = result[1];               //得到绳头的速度
54          connectionPos = connectionPos.add(connectionVel.multiConstant(dt));
                                                     //计算绳头的位置
55          if(connectionPos.y<groundHeight){        //保证绳头在地面的高度之上
56              connectionPos.y=groundHeight;
57              connectionVel.y=0;
58          }
59          head.pos=connectionPos;                  //设置绳头的位置
60          head.vel=connectionVel;                  //设置绳头的速度
61      }
62      public Vector3[] calCentripetalForceAndVel(){   //模拟产生绳头作用力与速度的方法
63          Mass head = massList.get(0);             //获取绳头质点
64          Vector3 center = new Vector3(0,head.pos.y,-2);//绕 y 轴旋转
65          Vector3 forceDir = center.add(head.pos.multiConstant(-1)).normal();
66          Vector3 force = forceDir.multiConstant(head.m);
                                                     //向心力 F=m*v*v/r  此处令 v=1,r=1;
67          Vector3 velDir = Vector3.yRotate(
68                  Math.toRadians(90), new double[]{forceDir.x,forceDir.y,
                    forceDir.z,1});
69          Vector3 vel = velDir.multiConstant(10);//速度大小为 10
70          if(head.pos.y<1.5f){
71              vel.y=1;                             //垂直速度为 1
72          }
73          return new Vector3[]{force,vel};         //返回绳头作用力和速度
74  }}
```

• 第7-23行为该类的构造器，其各个参数的含义分别为质点的数量、质点的质量、重力、地面反作用力系数、地面摩擦系数、地面缓冲系数、地面高度、空气阻力系数、绳头速度、绳头位置。在构造器中对各个力的系数、所有的质点和所有的弹簧进行了初始化。

• 第24-47行为计算绳子中各质点和弹簧受力情况的solve方法，各种力都是按照上一小节介绍的公式来进行计算的。

• 第48-61行为模拟绳子运动的方法，通过调用父类的模拟方法，对每个质点的运动进行了模拟。同时在一轮模拟结束后计算出假想的施加与绳子一端的作用力及此端质点的速度。

• 第62-74行为计算施加于绳子一端使绳子绕*Y*轴螺旋上升的假想力的calCentripetalForceAndVel方法，读者有兴趣的话可以修改此方法中的算法以牵引绳子沿不同的轨迹运动。

（5）最后需要给出的是在MySurfaceView类中绘制一帧画面的onDrawFrame方法，其代码如下。

代码位置：见随书光盘中源代码/第16章/Sample16_6/com/bn/Sample16_6目录下的MySurfaceView.java。

```
public void onDrawFrame(GL10 gl) {                        //绘制一帧画面的方法
    GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT | GLES20.GL_COLOR_BUFFER_BIT);
    MatrixState.pushMatrix();                             //保护现场
    for(int i=0;i<rope.massCount-1;i++){                  //循环遍历质点列表
        Mass mass1 = rope.massList.get(i);                //获取质点对象1
        Mass mass2 = rope.massList.get(i+1);              //获取质点对象2
        //计算出从质点1位置到质点2位置的方向向量
        Vector3 stickVector = mass1.pos.add(mass2.pos.multiConstant(-1));
        MatrixState.pushMatrix();                         //保护现场
        MatrixState.translate(mass2.pos.x, mass2.pos.y, mass2.pos.z);
                                                          //平移坐标系到质点2位置
            //将坐标系旋转，使得X轴与stickVector向量平行
            Vector3.moveXToSomeVector(new double[]{stickVector.x,stickVector.y,
            stickVector.z});
            stick.drawSelf(springLength/2,0,textureId); //绘制表示一截绳子的圆柱
            MatrixState.popMatrix();                        //恢复现场
      }
    MatrixState.popMatrix();                                //恢复现场
  }
```

说明

该方法的主要功能为根据上一小节介绍的绘制绳子的原理，每两个质点间用一根圆柱表示，最终绘制出整根绳子。

16.5　本章小结

本章主要向读者介绍了一些开发3D应用程序中经常会用到的碰撞检测以及物理计算方面的知识，包括AABB包围盒、粒子系统、弹簧质点系统等。读者掌握了本章的知识后，采用OpenGL ES 2.0模拟现实世界的能力将大大增强。

第17章

游戏的心脏——物理引擎

玩过极品飞车、都市赛车等虚拟现实类游戏的读者都会有这样的感觉，这些游戏的巨大吸引力不单来自于绚丽的画面，还来自于其真实的碰撞、飞跃等物理仿真效果。如果仅有绚丽的画面而对现实世界物理效果的模拟不真实，那么游戏的吸引力也将大打折扣。

前面的第16章简单介绍了一些与3D游戏开发相关的物理知识，但如果所有的复杂物理仿真效果都直接由游戏开发人员根据物理模型实现，开发的难度和成本将大大增加。因此，本章将向读者介绍实际游戏开发中经常使用的用于模拟现实世界物理效果的一种方式——物理引擎。

17.1　物理引擎很重要

有些读者可能对物理引擎已经有了简单的了解，但很多读者可能对其不是很熟悉。因此本节将对物理引擎进行一下简要的介绍，主要包括什么是物理引擎以及一些常见的物理引擎两部分内容。

17.1.1　什么是物理引擎

物理引擎通过给物体赋予真实的物理属性来模拟物体的运动，包括碰撞、移动、旋转等。并不是所有的游戏都必须使用独立的物理引擎，一些简单游戏中的物理效果模拟可以通过自行开发碰撞检测以及实现力学公式来完成对刚体或质点的模拟，就像前面第16章介绍的一样。

当游戏中需要实现比较复杂的刚体碰撞、滚动或者弹跳时，通过全部自行编程的方式实现就非常困难，成本也很高。遇到这种情况时就可以使用独立的物理引擎来模拟物体的运动，使用物理引擎不仅可以得到更加真实的效果，对于开发人员来说也比自行开发要耗时短、效率高。

一款好的物理引擎不仅可以帮助实现碰撞检测、力学模拟，而且还会提供很多关节的实现，如：铰链关节、滑动关节、六自由度关节等。通过各种关节的组合可以实现各种复杂的机械结构，详细内容在后继部分进行介绍。更高级的物理引擎不但可以提供刚体的模拟，甚至还可以提供软体及流体的模拟，这些都能大大提升游戏的真实感和吸引力。

17.1.2　常见的物理引擎

市面上存在的物理引擎数量是很多的，著名的物理引擎有Havok、Bullet、PhysX、ODE以及Box2D等。其中ODE、Bullet、Box2D等是开源的物理引擎，而PhysX的前身是Novodex，当被Ageia收购之后改名为PhysX，是一款可以免费用于非商业用途的引擎，商业用途及源代码需要付 费，Havok在许可方面也是如此。上述几种物理引擎的基本信息如表17-1所列。

表17-1　知名物理引擎的基本信息

物理引擎名称	Havok	PhysX	Bullet	Box2D	ODE
持有公司/人员	Intel	Nvidia	AMD	Erin Catto	Russell Smith
是否开源	否	否	是	是	是
是否支持 C/C++	是	是	是	是	是
最新版本	2011.2	3.0	2.79	2.2.1	0.11.1
文档情况	详细	详细	尚可	尚可	一般
支持的维数	3D	3D	3D	2D	3D

提示

目前世界3大3D物理引擎为Havok、PhysX和Bullet。另外2008年2月4日，NVIDIA成功收购AGEIA。此后凡是支持CUDA技术的显卡，就可以启动硬件PhysX加速，这可以大大提升游戏的实时性能。

- Havok。

Hovok成立于1998年，总部位于都柏林。在2000年游戏开发者大会上发布了Havok 1.0，最新版本为Havok2011.2，该引擎基于C/C++。2007年9月，Intel宣布成功收购Havok。之后，Intel宣布Havok引擎开放源代码并允许游戏开发人员免费用于非商业用途。

由于Havok的开放性和不依赖特定硬件的特点，很多大型游戏均使用Havok引擎。其中著名的有波斯王子：遗忘之沙、辐射3等，效果分别如图17-1与图17-2所示。



图17-1　波斯王子：遗忘之沙

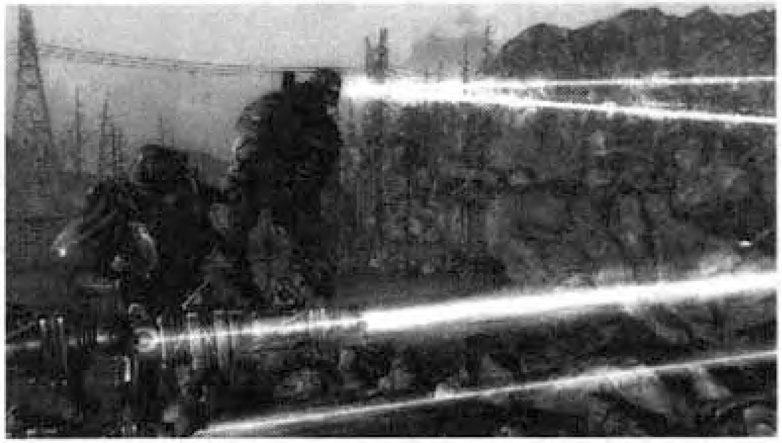


图17-2　辐射3

> **提示**
>
> 2011年3月6日，Intel宣布Havok引擎开始支持Android。并且已经与索爱达成合作关系，Havok将为索爱的Xperia Play游戏手机进行优化以便用于智能手机游戏的开发。2011年11月7日，Intel宣布，与Autodesk达成协议，Havok将提供新版本的视觉引擎，其中集成了Autodesk的Scaleform和Beast中间件。

- PhysX。

PhysX不仅可以由CPU计算，而且其程序本身在设计上可以使用独立的浮点处理器来计算。正是由于这个原因，其可以非常轻松地完成像流体力学那样计算量非常大的物理模拟计算。该引擎可以在Windows、Linux、Xbox360、Playstation3、Mac等多种平台上运行。

到目前为止在X86-PC平台上，共有200多种游戏采用PhysX引擎。被大家熟知的有蝙蝠侠：阿卡姆疯人院、幽灵行动等，其效果分别如图17-3和图17-4所示。



图17-3　蝙蝠侠：阿卡姆疯人院

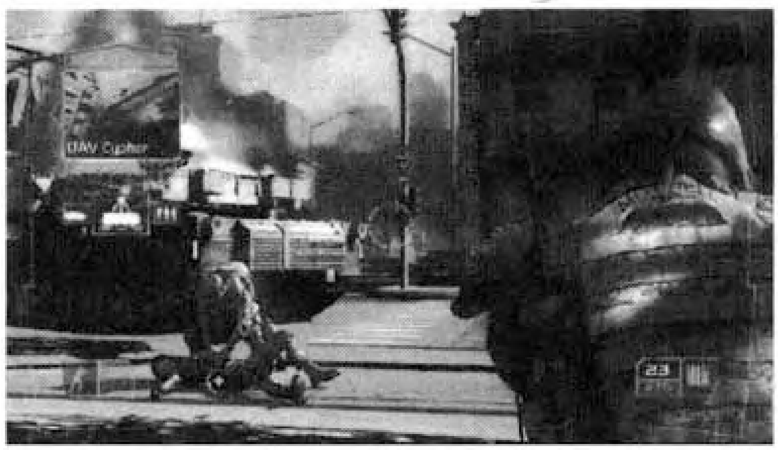


图17-4　幽灵行动

提示

2011年，NVIDIA发布了最新的SDK3.2Beta-1，这个新的版本提供了对PhysX 3下一代SDK初步的公共访问接口，并且对很多特性进行了加强和改进。

- Bullet。

Bullet是一款开源的3D物理引擎，是AMD开放物理计划的成员之一。同时其也是一个跨平台的物理引擎，支持Windows、Linux、MAC、Playstation3、XBOX360以及Nintendo Wii等主流平台。使用Bullet物理引擎开发的游戏主要有玩具总动员3等，其效果如图17-5所示。同时

使用该引擎制作的电影也不乏大作，如闪电狗就是迪斯尼使用的该引擎制作的3D动画电影，其效果如图17-6所示。



图17-5　玩具总动员3



图17-6　闪电狗

提示

Bullet的版本已经更新到2.79，该版本主要是对之前版本的bug进行了修复，不过也有增添新的特性，如集成了一个用于凸面分解的库——HACD。

- ODE。

ODE（Open Dynamic Engine）是一款免费的具有工业品质的刚体动力学引擎，其可以非常好地仿真现实世界中物体的移动、旋转等，具有良好的可移植性，并且内置了碰撞检测系统。ODE目前可以支持球

窝、铰链、滑块、定轴、角电机和hinge-2等关节类型，还可以支持各种碰撞形式（如球面碰撞和平面碰撞）以及多个碰撞空间。

提示

ODE现在还没有正式发布1.0版本，最新版为0.11.1，还属于不成熟阶段，因此应用不如前几种广泛。

- Box2D。

Box2D是一款非常著名的2D物理引擎，主要用于2D刚体仿真，有C++、Flash和Java等版本，通过其能非常好地进行2D世界的物理效果模拟。Box2D最早是用可移植的C++开发的，后来随着需求的变化陆续推出了Java与Flash的版本。最近非常受欢迎的游戏《愤怒的小鸟》就是基于该引擎开发的，其效果如图17-7与图17-8所示。

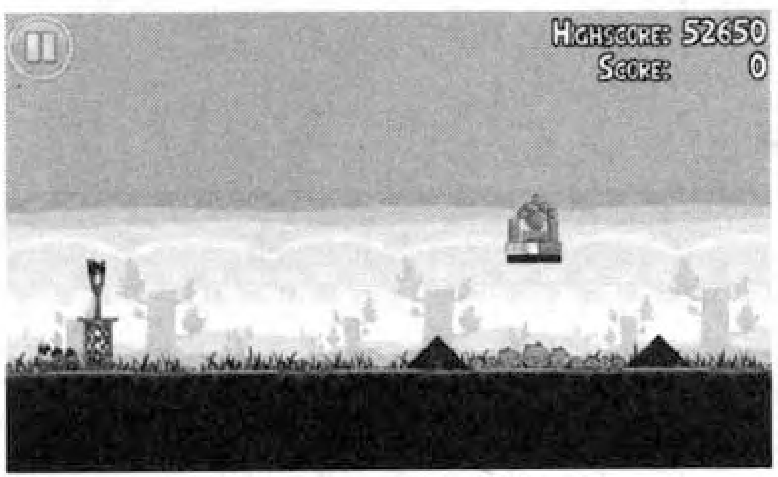


图17-7　游戏开始



图17-8　游戏进行中

17.2　JBullet物理引擎

JBullet是开源的物理引擎Bullet的Java版本，因此可以直接用于Android。该引擎能够根据开发人员设定的参数，如：重力、密度、摩擦系数和弹性系数等，自动地进行3D刚体物理运动的全方位模拟。

> **提示**
>
> 每种物理引擎都有其独特的一些基本概念，在学习开源物理引擎时，首先需要弄懂的就是这些基本概念。如果在基本概念不清楚的情况下去“深入”学习，只会使自己陷入泥沼中，并且越陷越深，难以有所成就。因此，本节首先带领读者复习一下物理学中的一些基本概念，然后介绍JBullet中的一些常用类与概念。

17.2.1　基本的物理学概念

游戏实际上是对真实世界的仿真，其中用到了许多物理学知识，如：密度、质量、质心、摩擦力、扭矩以及碰撞（恢复）系数等。因此，本小节将简要回顾一下大家中学阶段学习过的，用JBullet进行开发时经常碰到的一些物理学概念。

- 密度。

密度指的是单位体积的质量，符号为“p”，常用单位为kg/m^3 。密度是物质的一种基本特性，不随物体的质量、体积的改变而改变，同种物质的密度相同。

- 质量。

质量指的是物体中所含物质的量，即物体惯性的大小，国际单位是kg。同一物体的质量通常是一个常量，不随其位置、姿态的改变而变化。但是根据爱因斯坦的相对论，同一物体的质量会随着速度的变化而改变。由于只有运动速度接近光速才能感觉到这种变化，故在游戏中一般不考虑速度对质量的影响。

- 质心。

物体（或物体系）的质量中心，是研究物体（或物体系）机械运动的一个重要参考点。当作用力（或合力）通过该点时，物体只作平移而不发生转动；否则在发生平移的同时物体将绕该点转动。研究质心的运动时，可将物体的质量看作集中于质心。理论上，质心是对物体的质量分布用“加权平均法”求出的平均中心。

- 摩擦力。

当两个互相接触的物体要发生或者已经发生相对运动时，就会在接触面上产生一种阻碍该相对运动的力。这种力就称之为摩擦力，其基本情况如图17-9所示。

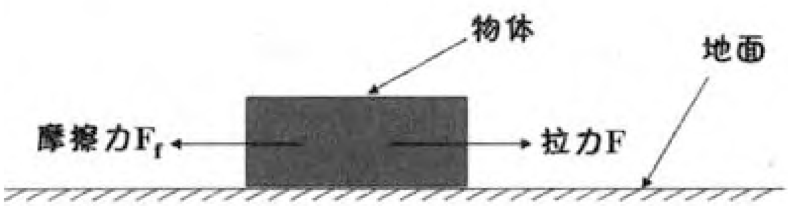


图17-9　摩擦力示意图

> **提示**
>
> 根据物体是否发生相对运动可以分为静摩擦力与滑动摩擦力，一般情况下静摩擦力大于滑动摩擦力。实际开发中可以进行简化，但若要模拟更加真实的效果就需要进行分别开发。

- 扭矩。

扭矩在物理学中就是力矩的大小，其等于力与力臂的乘积，国际单位是Nm（牛米）。在力臂不变的情况下，力越大，扭矩越大，基本情况如图17-10所示。

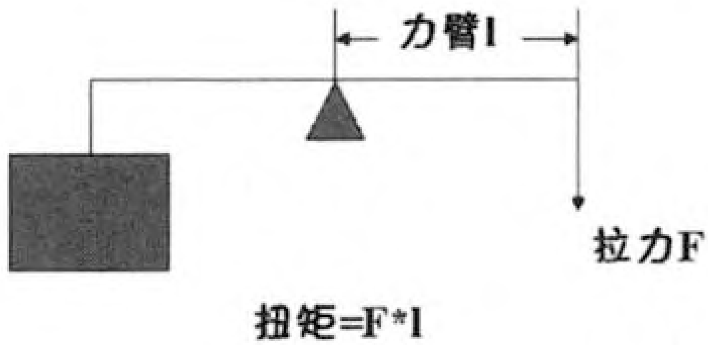


图17-10　扭矩示意图

- 恢复系数。

两物体碰撞后总动能与碰撞前总动能之比称之为恢复系数，其取值范围为0～1。如果恢复系数为1，则碰撞为完全弹性碰撞，满足机械能守恒；如果恢复系数小于1并且大于0，则为非完全弹性碰撞，不满足机械能守恒，这种情况是最常见的；如果恢复系数为0，则为完全非弹性碰撞，两个物体会黏在一起。几种基本情况如图17-11所示。

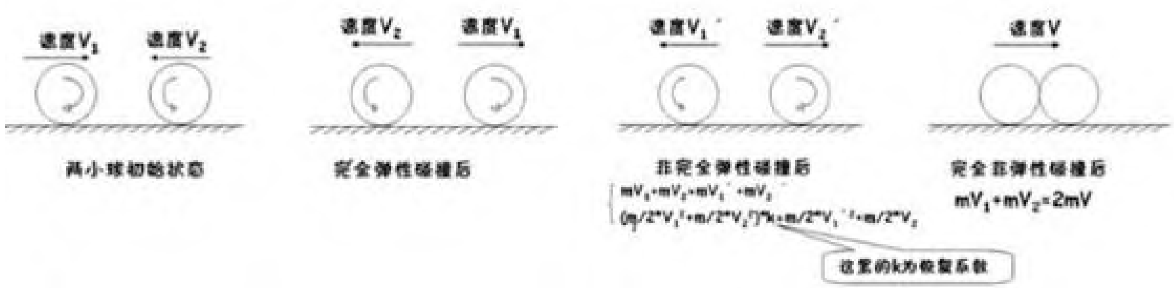


图17-11　各种碰撞情况

说明

图17-11中，两个小球的质量相等，小球的初速度V_1 大于V_2 。

17.2.2　JBullet中常用类的介绍

俗话说得好“磨刀不误砍柴工”，在学习新技术时，首先要弄清楚一些基本的概念，一些常用的API，这样深入学习起来才更有效率。因此，本小节主要介绍一些学习JBullet过程中必知必会的类及其对应的概念。

1．Vector3f类（三维向量类）

该类属于javax.vecmath包，继承自同一包下的Tuple3f（三元组）类。该类的使用频率非常高，其对象可以表示速度、点、力等向量（矢量），其基本功能如图17-12所示。

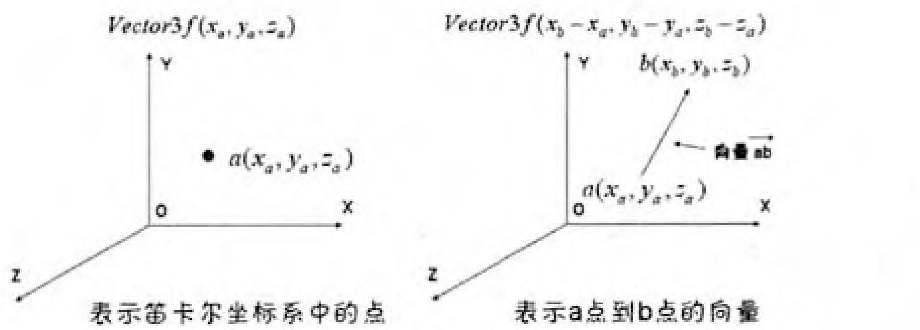


图17-12　Vector3f类对象功能示意图

从图17-12中可以看出，三维坐标系中的点和向量均可以用该类的对象来表示。该类的构造器与常用方法如表17-2所列。

表17-2　Vector3f类构造器与常用方法

方法或构造器签名	含　义	类型
public Vector3f()	创建一个三个维度分量初始值都为0的三维向量对象	构造器
public Vector3f(float[]f)	由一个长度为3的数组中元素的值创建三维向量对象	构造器
public Vector3f(Vector3f v)	由另一个三维向量对象3个维度的分量值创建新的三维向量对象，参数v代表提供3个维度分量值的三维向量对象	构造器

public Vector3f(float a, float b, float c)	由3个浮点数创建一个三维向量对象。参数a代表第一个维度的分量值，参数b代表第二个维度的分量值，参数c代表第三个维度的分量值	构造器
public float dot(Vector3f v)	与另一个三维向量进行点积操作，参数v代表要与当前向量进行点积的向量	方法
public void cross(Vector3f v1,Vector3f v2)	将两个三维向量进行叉积计算，结果存放于当前三维向量对象中。参数v1代表叉积的起始向量，参数v2代表叉积的终止向量	方法
public void add(Tuple3f v)	将两个向量相加，参数v代表与当前向量作向量加法的向量	方法
public void normalize()	将向量进行规格化，也就是向量的方向不变但长度（模）变为1	方法
public float length()	求向量的长度的方法，返回向量的长度值	方法
public void set(Tuple3f v)	设置当前向量的值，参数v代表用来设置当前向量的值的向量	方法

提示

从Vector3f类所属的包名中可以看出，其并不是属于JBullet特有的包。确实如此，由于JBullet本身是开源的，因此其也采用了其他的一些免费工具箱，Vector3f类就是来自于Java3D中的vecmath.jar这个jar包。

2．Transform类（变换类）

该类属于com.bulletphysics.linearmath包，继承自java.lang.Object类，用来表示刚体的变换，如平移、旋转等。该类构造器与常用方法及属性如表17-3所列。

表17-3　Transform类构造器与常用方法及属性

方法、属性或构造器签名	含　义	类型
public final Matrix3f basis	表示旋转变换的3×3矩阵	属性
public final Vector3f origin	表示平移变换的向量	属性
public Transform()	无参构造器	构造器
public Transform(Matrix3f mat)	参数mat表示传入的3×3矩阵，用来初始化表示旋转变换的3×3矩阵	构造器
public Transform(Matrix4f mat)	参数mat表示传入的4×4矩阵，用来初始化表示旋转变换的3×3矩阵以及表示平移变换的向量	构造器
public Transform(Transform tr)	参数tr表示传入的另一个Transform对象，用该对象的属性值初始化新建的Transform对象	构造器
public void transform(Vector3f v)	进行平移变换的方法，参数v代表平移变换对应的向量	方法
public void setIdentity()	将Transform对象设置为初始状态，即将旋转变换矩阵单位化，平移向量3个维度的分量归零	方法
public void mul(Transform tr)	将当前变换对象与另一个变换对象复合的方法，参数tr表示的是在当前变换的基础上附加的平移与旋转变换	方法
public void mul (Transform tr1, Transform tr2)	设置当前变换对象为另外两个变换复合的结果，参数tr1表示一个变换对象，参数tr2表示与tr1复合的另一个变换对象	方法
public void setRotation(Quat4f q)	设置当前变换对象的旋转变换数据，参数q表示存储旋转数据的四元数对象	方法
public Matrix4f getMatrix(Matrix4f	获取当前变换对象所存储的变换信息。参数out代表由旋转缩放矩阵和平移向量	方法

out)	所合成的4×4变换矩阵	

3．RigidBody类（刚体类）

该类属于com.bulletphysics.dynamics包，继承自com.bulletphysics.collision.dispatch包下的CollisionObject类，其对象用于存储刚体的一些属性信息，包括线速度、角速度、摩擦系数等。该类的构造器及常用方法如表17-4所列。

表17-4　RigidBody类的构造器和常用方法

构造器或方法签名	含　义	类型
public RigidBody (float mass, MotionState motionState, CollisionShape collisionShape)	创建一个刚体对象。参数mass代表刚体的质量，参数motionState为代表刚体运动状态对象，用于记录刚体的位置、姿态等信息，参数collisionShape表示用于碰撞计算的形状	构造器
public void setGravity (Vector3f acceleration)	设置刚体的重力加速度，参数acceleration代表重力加速度的值	方法
public void setRestitution(float restitution)	设置反弹系数，参数restitution为反弹系数的值	方法
public void setFriction(float friction)	设置摩擦系数，参数friction为摩擦系数的值	方法
public MotionState getMotionState()	获取存储刚体位置、姿态的MotionState对象	方法
public void clearForces()	清除刚体所有的受力	方法
public Vector3f	获取刚体的线速度，参数out代表保存	方法

getLinearVelocity (Vector3f out)	刚体线速度的向量	
public Vector3f getAngularVelocity (Vector3f out)	获取刚体转动的角速度，参数out表示记录刚体转动角速度的向量	方法
public void setLinearVelocity (Vector3f lin_vel)	设置刚体的线速度，参数lin_vel表示要设置的线速度	方法
public void setAngularVelocity (Vector3f ang_vel)	设置刚体的角速度，参数ang_vel表示要设置的角速度	方法

4．CollisionShape类（碰撞形状抽象类）

该类属于com.bulletphysics.collision.shapes包，其为继承自java.lang.Object的抽象类，是所有碰撞形状具体类的根类，也就是说后面将要介绍的所有的形状类都直接或间接继承自此类。碰撞形状主要用来描述对应刚体所模拟物体的形状，该类中的常用方法如表17-5所列。

表17-5　CollisionShape类的常用方法

方法签名	含　义	类型
public void getAabb(Transform t, Vector3f aabbMin, Vector3f aabbMax)	获取AABB包围盒边界参数的方法。参数t代表要执行的变换，参数aabbMin表示存储边界3个最小值的向量，参数aabbMax表示存储边界3个最大值的向量	方法
public BroadphaseNativeType getShapeType()	获取当前碰撞形状类别的方法。返回值为枚举类型BroadphaseNativeType，具体包括如 BOX_SHAPE_PROXYTYPE、CONE_SHAPE_PROXYTYPE、	方法

	TRIANGLE_SHAPE_PROXYTYPE、CAPSULE_SHAPE_PROXYTYPE等	
public boolean isConvex()	判断该当前形状是否是凸面形状，若是则返回true,否则返回false	方法
public boolean isConcave()	判断当前形状是否是凹面形状，若是则返回true,否则返回false	方法
public boolean isCompound()	判断当前形状是否是复合形状，若是则返回true,否则返回false	方法
public boolean isPolyhedral()	判断当前形状是否是多面体，若是则返回true,否则返回false	方法
public void setLocalScaling (Vector3f scaling)	设置此形状3个轴方向的缩放比，可以用来对形状进行变形操作。参数scaling3个方向的分量分别代表3个轴方向的缩放比	方法
public abstract Vector3f getLocalScaling (Vector3f out)	获取此形状3个轴方向的缩放比，参数out为获取的3个轴方向的缩放比	方法
public void calculateLocalInertia (float mass, Vector3f inertia)	计算形状惯性的方法。参数mass代表刚体的质量，参数inertia为用来保存惯性计算结果的向量	方法

5．BoxShape类（长方体盒碰撞形状）

该类属于com.bulletphysics.collision.shapes包，直接继承自同一包下的PolyhedralConvexShape类，间接继承自前面介绍过的CollisionShape类。该形状可用于盒子、箱子等规则的物体，对于一些不规则的物体（如茶壶、兔子等）其部分部位（如头部）可以采用盒子来模拟，其构造器如表17-6所列。

表17-6　BoxShape类构造器和常用方法

构造器签名	含　义	类型
public BoxShape (Vector3f boxHalfExtents)	创建一个长方体盒碰撞形状对象。参数boxHalfExtents 3个维度的分量值分表代表长方体盒长、宽、高的半边长	构造器

6．StaticPlaneShape类（静态平面形状）

该类属于com.bulletphysics.collision.shapes包，继承自同一包下的ConcaveShape类。该类的对象表示静态的平面，如地面、屋顶等，其构造器与常用方法如表17-7所列。

表17-7　StaticPlaneShape类构造器和常用方法

构造器或方法签名	含　义	类型
public StaticPlaneShape (Vector3f planeNormal, float planeConstant)	创建一个静态平面碰撞形状对象。参数planeNormal代表平面的法向量。参数planeConstant代表平面上的任意一点	构造器
public Vector3f getPlaneNormal (Vector3fout)	获取平面法向量的方法，参数out代表保存平面法向量的向量	方法

提示

在满足程序视觉效果需求的情况下，这样的近似模拟比直接使用精确的形状进行碰撞计算速度要快很多，是一种常用的技术手段。

7．SphereShape类（球体形状）

该类属于com.bulletphysics.collision.shapes包，继承自同一包下的ConvexInternalShape类，其对象表示一个球形。球形的物体可以选择该类对象作为碰撞形状，如篮球、足球等，其构造器及常用的方法如表17-8所列。

表17-8　SphereShape类构造器和常用方法

构造器或方法签名	含　义	类型
public SphereShape(float radius)	创建一个球形碰撞形状对象，参数rdins代表球的半径	构造器
public float getRadius()	获取球的半径	方法

8．CylinderShape类（圆柱形状）

该类属于com.bulletphysics.collision.shapes包，继承自同一包下的BoxShape类，其对象表示一个圆柱形状。很多圆柱形状的物体，如很长的杆、石柱、金币等，都可以采用该类对象作为碰撞形状，其构造器如表17-9所列。

表17-9　CylinderShape类构造器和常用方法

构造器签名	含　义	类型
public CylinderShape(Vector3f halfExtents)	创建一个侧面围绕Y轴的圆柱形碰撞形状对象。参数halfExtents为表示圆柱尺寸的三元组，其中第一和第三维度的分量表示圆柱截面（椭圆）的长短半径，第二维度分量为圆柱的长度	构造器

提示

请读者特别注意的是，采用圆柱形状进行碰撞计算需要很大的计算量，这可能会导致游戏的帧速率低下。因此在能够采用胶囊碰撞形状的情况下应该尽量避免使用圆柱碰撞形状，关于胶囊的具体情况会在后面的部分进行介绍。

提示

CylinderShape类的对象表示的是质心在原点，包围y 轴的圆柱。另外还有CylinderShapeX和CylinderShapeZ两个类，其对象分别表示质心在原点，包围x 轴和包围z 轴的圆柱。CylinderShapeX和CylinderShapeZ这两个类本质上与CylinderShape类没有区别，只是初始时圆柱的姿态不同，因此3个类的构造器非常类似，这里不再赘述。

9．CapsuleShape类（胶囊形状）

该类属于com.bulletphysics.collision.shapes包，继承自同一包下的ConvexInternalShape类，其对象表示一个胶囊形状。比较细长的圆柱形物体一般不直接采用圆柱形状而是采用胶囊形状，如旗杆、铅笔等。

提示

之所以愿意在开发中尽量使用胶囊是因为胶囊在碰撞计算中相比于圆柱有很大的速度优势，这是因为圆柱的两端是平面，与侧面的柱面正交，比较难于计算。而胶囊柱面的两端是半球面，计算起来就简单多了（如图17-13所示）。由于本章着重于介绍JBullet的使用，而不是如何开发物理引擎，因此计算的细节略去，有兴趣的读者可以参考其他的技术资料。

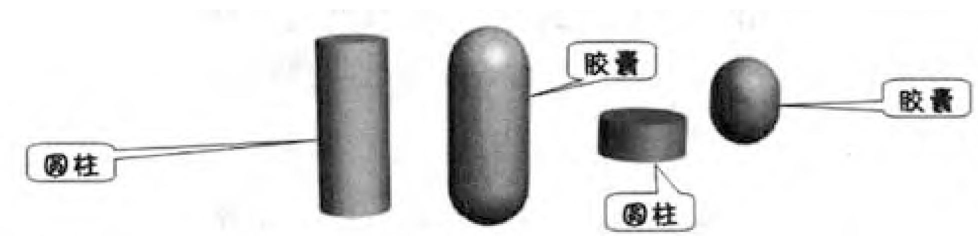


图17-13　胶囊与圆柱的比较

另外要注意的是，并不是在所有的情况下都可以采用胶囊代替圆柱的。例如，开发推金币游戏时，游戏中的金币（很扁的圆柱）就不适合采用胶囊，否则效果会很不真实，这一点从图17-13中可以很明显地看出来。CapsuleShape类构造器及常用的方法如表17-10所列。

表17-10　CapsuleShape类构造器和常用方法

构造器和方法签名	含　义	类型
public CapsuleShape(float radius, float height)	创建一个胶囊碰撞形状对象。参数radius代表两端球面的半径，参数height代表胶囊中间圆柱部分的长度	构造器
public float getRadius()	获取胶囊截面（圆柱部分）的半径	方法
public float getHalfHeight()	获取中间圆柱部分高度值的一半	方法

提示

CapsuleShape类对象表示的是质心在原点，包围少轴的胶囊，另外还有CapsuleShapeX和CapsuleShapeZ两个类，其对象分别表示质心在原点，包围x轴和包围z轴的胶囊。CapsuleShapeX和CapsuleShapeZ这两个类本质上与CapsuleShape类没有区别，只是初始时胶囊的姿态不同，因此3个类的构造器与方法基本相同，这里不再赘述。

10．CompoundShape类（复合形状）

该类属于com.bulletphysics.collision.shapes包，继承自同一包下的CollisionShape类，其对象表
示一个复合形状。由于现实世界中的物体形状纷繁复杂，有时用单一的形状描述某一物体时有很大偏差，这样仿真的效果也很不真实。

此时，应采用的策略是将形状复杂的物体拆分为若干组成部分，每一部分再由某单一形状描述。然后将各部分的形状复合成一个整体，用来描述整个物体，这样就可以实现较为真实的效果。这时就需要用到复合形状——CompoundShape类，其常用方法如表17-11所列。

表17-11　CompoundShape类的常用方法

方法签名	含　义
public void addChildShape (Transform localTransform, CollisionShape shape)	向组合形状中添加子形状的方法。参数localTransform代表添加的子形状的原始姿态、位置变换，参数shape代表添加的子形状
public void removeChildShape (CollisionShape shape)	从组合形状中删除指定的子形状，参数shape代表要删除的子形状
public int getNumChildShapes()	返回当前组合形状中子形状的数量
public CollisionShape getChildShape(int index)	返回组合形状中指定索引编号的子形状，参数index为指定的索引编号
public Transform getChildTransform(int index, Transform out)	返回组合形状中指定索引编号子形状的原始姿态、位置变换。参数index为指定的索引编号，参数out为获取的原始姿态、位置变换

11．BvhTriangleMeshShape类（层次包围结构三角形组形状）

该类属于com.bulletphysics.collision.shapes包，继承自同一包下的TriangleMeshShape类，其对象表示一个由多个三角形组成的立体形状。BvhTriangleMeshShape类对象主要用来表示形状比较复杂的静态物体，如地形、建筑物、山岗等。

此类专门针对静止的物体做了性能优化，在实际场景的开发中非常有用，其常用的几个构造器如表17-12所列。

表17-12　BvhTriangleMeshShape类的构造器

构造器签名	含　义
public BvhTriangleMeshShape()	无参构造器
public BvhTriangleMeshShape (StridingMeshInterface meshInterface, boolean useQuantizedAabbCompression)	有参构造器。参数meshInterface代表创建形状时需要用到的三角形组中各个三角形的顶点数据，参数useQuantizedAabbCompression表示是否使用量化的AABB盒压缩
public BvhTriangleMeshShape (StridingMeshInterface meshInterface, boolean useQuantizedAabbCompression, boolean buildBvh)	有参构造器。参数meshInterface代表创建形状时需要用到的三角形组中各个三角形的顶点数据，参数useQuantizedAabbCompression表示是否使用量化的AABB盒压缩，参数buildBvh代表是否使用层次包围结构

12．GlmpactMeshShape类（Glmpact三角网状形状）

该类属于com.bulletphysics.extras.gimpact包，继承自同一包下的GImpactShapeInterface抽象类。 GImpactShapeInterface抽象类直接继承自com.bulletphysics.collision.shapes包下的ConcaveShape,间接继承自CollisionShape类。

该类的对象表示的也是由多个三角形组成的立体形状，与前面介绍的BvhTriangleMeshShape类不同的是，GimpactMeshShape类的对象

主要是用来描述形状比较复杂的运动物体，如运动的兔子、圆环、茶壶等。GimpactMeshShape类的构造器和常用方法如表17-13所示。

表17-13　GimpactMeshShape类构造器和常用方法

构造器和方法签名	含　义	类型
public GimpactMeshShape (StridingMeshInterface meshInterface)	有参构造器，参数meshInterface代表创建形状时需要用到的三角形组中各个三角形的顶点数据	构造器
public int getMeshPartCount()	返回此形状组成部分的数量	方法
public GImpactMeshShapePart getMeshPart(int index)	返回此形状指定索引编号的组成部分，参数index为给定的索引编号	方法

13．TriangleIndexVertexArray类（三角形组顶点数据数组）

该类属于com.bulletphysics.collision.shapes包，继承自同一包下的StridingMeshInterface抽象类。该类的对象用来封装一个由三角形组成的立体形状中各个顶点的数据，数据的组织方式采用的是前面第5章中介绍的索引法，其构造器和常用方法如表17-14所示。

表17-14　TriangleIndexVertexArray类的构造器和方法

构造器和方法签名	含　义	类型
public TriangleIndex VertexArray()	无参构造器，创建一个三角形组顶点数据数组对象	构造器
public TriangleIndexVertexArray (int numTriangles, ByteBuffer	有参构造器。参数numTriangles为三角形组中三角形的个数，参数triangleIndexBase为索引数据缓冲，参数triangleIndexStride为	构造器

triangleIndexBase, int triangleIndexStride, int numVertices, ByteBuffer vertexBase, int vertexStride)	索引数据间隔，参数numVertices为总共的顶点个数，参数vertexBase为顶点数据缓冲，参数vertexStride为顶点数据间隔	
public void addIndexedMesh (IndexedMesh mesh)	向三角形组顶点数据数组中添加顶点数据，参数mesh为封装了要添加的顶点数据的对象，数据采用索引法组织。采用此方法时顶点索引数据必须是整型的，即对应ScalarType.INTEGER的类型	方法
public void addIndexedMesh (IndexedMesh mesh, ScalarType indexType)	向三角形组顶点数据数组中添加顶点数据，参数mesh为封装了要添加的顶点数据的对象，数据采用索引法组织。采用此方法时顶点索引数据的类型可以有5种选择，包括：FLOAT、DOUBLE、INTEGER、SHORT、FIXEDPOINT88。上述5种选择都是ScalarType枚举类型下的枚举值	方法
public int getNumSubParts()	获取三角形组中组成部分的数量，及此三角形组中包含了几个IndexedMesh	方法

提示

IndexedMesh类非常简单，其继承自Object类，里面仅包含了7个成员变量，没有任何自有的方法。这7个成员变量的前6个与TriangleIndexVertexArray类有参构造器的6个参数相同，第7个为索引数据的类型，是ScalarType枚举类型的。

本小节对JBullet中常用类的API进行了简要的介绍，但并不够全面。幸运的是，JBullet是开源的项目，需要更详细信息的读者可以到网上下载JBullet的源代码进行深入的学习，笔者也是这么做的。为了

使读者能够更好地掌握本小节介绍的多个形状类之间的继承关系，下面给出从CollisionShape类衍生出的与本小节介绍的各个形状类相关的继承树，如图17-14所示。

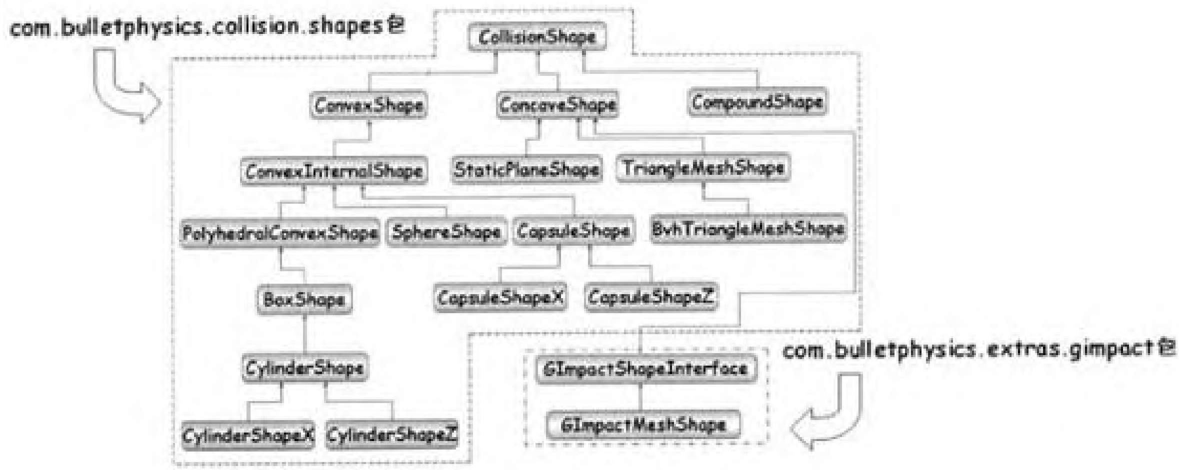


图17-14　碰撞形状的继承树

说明

从图17-14中可以看出，所有的碰撞形状都直接或间接继承自CollisionShape类。另外，中间的一些类由于很少直接使用，因此本小节没有详细介绍，有兴趣的读者可以参考JBullet的源代码。

17.3　箱子相互碰撞的案例

上一节主要介绍了JBullet的一些基本概念及API，但仅学习概念和API是远远不够的。因此从本节开始，笔者将通过一系列的小案例来向读者介绍JBullet的具体使用，本节将介绍的是一个 箱子相互碰撞的案例Sample17_1。

17.3.1　案例运行效果及准备工作

本节给出的是一个箱子相互碰撞的案例，其运行效果如图17-15、图17-16、图17-17与图17-18这4幅图所示。

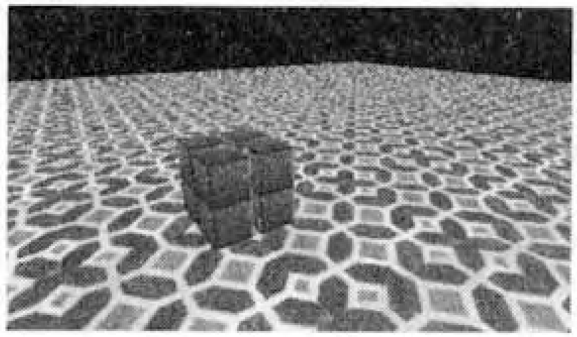


图17-15　案例初始状态

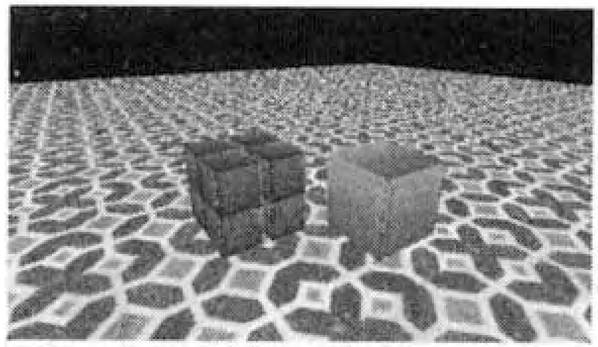


图17-16　抛出箱子

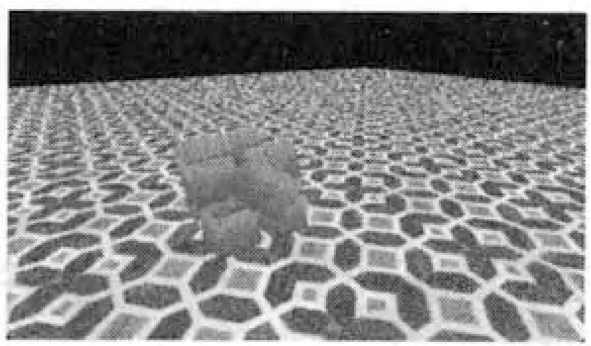


图17-17　开始碰撞

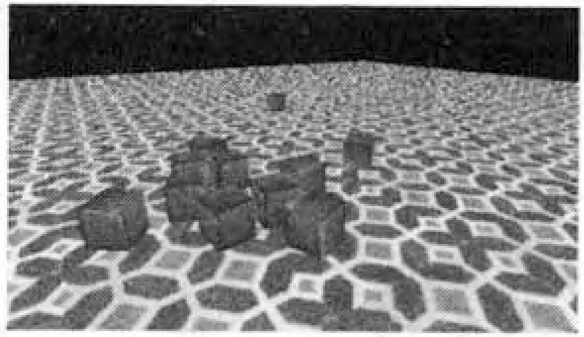


图17-18　碰撞结果

- 图17-15为案例运行后的初始状态，地面上摆放了8个静止的立方体木箱。
- 图17-16为手指触控屏幕后抛出一个木箱的情况。
- 图17-17为木箱开始相互碰撞的情况。
- 图17-18为抛出了多个箱子碰撞后散落各处的情况。

提示

为了便于观察，本案例中对静止的和运动的木箱采用了不同颜色的纹理。其中静止的木箱采用的是偏红色的纹理，运动的木箱采用的是偏蓝色的纹理。由于插图采用灰度印刷，因此这一点可能看得不是很清楚，此时请读者自行采用真机运行本案例进行观察即可。

了解了本案例的运行效果后，就应该进行开发前的准备工作了。由于本案例中采用了开源的物理引擎JBullet，因此需要用到两个第三方jar包，包括jbullet.jar与vecmath.jar。

- jbullet.jar中包含了JBullet物理引擎的各个实现类。
- vecmath.jar是来自于Java3D中的免费工具包，主要是包含了一些实现向量数学的类。

这两个jar包都可以从JBullet的官方网站“http：//jbullet.advel.cz/”上下载到，同时为了读者学习方便，随书光盘中也已经将这两个jar包集成到项目的lib目录下了，读者直接将项目从光盘导入Eclipse即可使用。

提示

如果读者开发自己的项目只需要将这两个jar包添加到项目中即可，若读者不清楚如何在Eclipse中为项目添加jar包可以查阅其他相关资料，本书由于篇幅所限不再赘述。

17.3.2　案例的基本框架结构

详细介绍本节案例中的各个重要类之前，首先应该对本节案例的框架结构有一个大致的了解，如图17-19所示。



图17-19　案例基本框架结构

- GL_Demo类为本案例的主控制类，也就是本案例中的Activity。案例开始运行时首先调用 此类中的onCreate方法，在该方法中首先执行全屏显示，强制横屏等操作，然后生成MySurfaceView类的对象，并进行必要的设置，最后跳转到MySurfaceView界面。
- MySurfaceView类是负责3D场景渲染的核心类，其负责对物理世界及其中的碰撞形状进行初始化，并创建初始场景中静止的箱子，同时处理屏幕触控事件的方法也在其中。
- Constant类为本案例中的常量类，其主要目的是将本案例中需要使用的一些常量集中管理。
- TexFloor为用于表示水平地面的类，其中不但包含了与渲染相关的代码还包含了与物理效果模拟相关的代码。
- TextureRect为纹理矩形类，其对象用于渲染呈现立方体木箱的一个面。
- TexCube为用于表示木箱的类，其中不但包含了与渲染相关的代码还包含了与物理效果模拟相关的代码。
- ShaderManager类包含若干静态方法，主要功能为对案例中用到的着色器进行统一地创建、存储，方便着色器程序的复用和管理。

提示

MatrixState及ShaderUtil类与本书前面很多案例中的基本一致，这里不再赘述。

17.3.3 常量类——Constant

正式开发案例中的各个具体功能类之前应该首先开发出常量类Constant，在其中给出各个功能类所需要使用的各种常量的定义，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_1/src/com.bn.Sample17_1目录下的Constant.java。

```
package com.bn.Sample17_1                              //声明包
 public class Constant{
   final static float UNIT_SIZE=0.5f;                  //基本单元尺寸
   final static float TIME_STEP=1.0f/60;               //物理模拟的频率
   final static int MAX_SUB_STEPS=5;                   //最大迭代子步数
   final static float EYE_X=-5;                        //摄像机的x坐标
   final static float EYE_Y=4;                         //摄像机的y坐标
   final static float EYE_Z=5;                         //摄像机的z坐标
   final static float TARGET_X=0;                      //摄像机目标点的x坐标
   final static float TARGET_Y=0;                      //摄像机目标点的y坐标
   final static float TARGET_Z=0;                      //摄像机目标点的z坐标
 }
```

提示

将应用中需要用到的常量集中到一起进行管理是一个很好的习惯，这样做有助于显著提高代码的可维护性，降低开发中不必要的维护成本。

17.3.4 3D场景渲染类——MySurfaceView

完成了常量类Constant的开发后，就可以开发实施整体3D场景渲染的MySurfaceView类了。其完成的工作很多，主要包括：3D场景的搭建、碰撞形状的初始化、物理世界的初始化、模拟线程的创建与启动、触控事件的处理等。

由于3D场景的搭建与本书前面很多的案例并无太大差异，因此这里仅着重介绍MySurfaceView类中与使用JBullet物理引擎密切相关的部分，具体内容如下。

（1）首先给出的是对物理世界进行初始化的initWorld方法，其具体代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_1/src/com.bn.Sample17_1目录下的MySurfaceView.java。

```
1 public void initWorld(){                                    //初始化物理世界的方法
2   //创建碰撞检测配置信息对象
3   CollisionConfiguration collisionConfiguration = new DefaultCollisionConfiguration();
4   //创建碰撞检测算法分配者对象，用来扫描所有的碰撞检测对，确定使用何种检测策略
5   CollisionDispatcher dispatcher = new CollisionDispatcher(collisionConfiguration);
6   Vector3f worldAabbMin = new Vector3f(-10000, -10000, -10000);//设置整个物理世界的边界最小值
7   Vector3f worldAabbMax = new Vector3f(10000, 10000, 10000);//设置整个物理世界的边界最大值
8   int maxProxies = 1024;                                    //最大代理数量
9   //创建碰撞检测粗测阶段的加速算法对象
10    AxisSweep3  overlappingPairCache  =new  AxisSweep3(worldAabbMin,  worldAabbMax,
maxProxies);
11  //创建推动约束解决者对象
12  SequentialImpulseConstraintSolver solver = new SequentialImpulseConstraintSolver();
13  dynamicsWorld = new DiscreteDynamicsWorld                 //创建物理世界对象
14              (dispatcher, overlappingPairCache, solver,collisionConfiguration);
15  dynamicsWorld.setGravity(new Vector3f(0, -10, 0));       //设置重力加速度
16  boxShape=new BoxShape                                     //创建共用的立方体碰撞形状
17  (new Vector3f(Constant.UNIT_SIZE,Constant.UNIT_SIZE,Constant.UNIT_SIZE));
18  planeShape=new StaticPlaneShape(new Vector3f(0, 1, 0), 0);//创建共用的平面碰撞形状
19 }
```

- 第6-7行创建了两个三维向量，用来指定物理世界的边界，其中worldAabbMin代表边界最小值，worldAabbMax代表边界最大值。实际物理模拟时，当物体超出边界后将不再进行模拟，以提高效率。
- 第10行创建粗测阶段的加速算法对象，在构造器中传入边界信息和最大代理数。
- 第16-17行创建了共用的立方体碰撞形状以及平面碰撞形状，在需要时同样碰撞形状的刚体可以共用同一个碰撞形状对象，提高资源利用率。本案例中的所有箱子就是共用的同一个立方体碰撞形状，这一点在后面可以看到。

（2）了解了对物理世界进行初始化的initWorld方法后，下面给出的是定时进行物理模拟操作的线程，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_1/src/com.bn.Sample17_1目录下的MySurfaceView.java。

```
1  public void onSurfaceCreated(GL10 gl, EGLConfig config) {
2      ……//此处省略了部分不重要的代码，读者可以自行查阅随书光盘中的源代码。
3      new Thread(){                                          //通过匿名内部类创建线程对象
4        public void run(){                                   //重写线程中执行任务的 run 方法
5          while(true){                                       //模拟循环
6            try{
7              synchronized(tcaForAdd){                       //锁定待添加箱子集合
8                synchronized(tca){                           //锁定总箱子集合
9                  for(TexCube tc:tcaForAdd){tca.add(tc);}    //向总箱子集合中添加箱子
10               }
11               tcaForAdd.clear();                           //将待添加箱子集合清空
12             }
13             dynamicsWorld.stepSimulation(TIME_STEP, MAX_SUB_STEPS);//执行模拟
14             Thread.sleep(20);                              //当前线程睡眠 20ms
15           }catch (Exception e){ e.printStackTrace();}      //若有异常则打印异常栈
16   }}}.start();                                             //启动线程
17 }
```

- 第7-12行为将待添加列表中的箱子添加到总的箱子列表中，并清空待添加箱子列表。之所以要采用同步代码是为了避免不同线程同时对同一集合进行增删操作可能带来的问题。
- 第13行是本线程的核心任务，调用stepSimulation方法执行物理模拟操作。其中MAX_SUB_STEPS为模拟运算时允许的最大迭代次数，其越大模拟效果越好，但运算速度也会越慢。因此，实际开发中应该恰当调整此值，合理平衡速度与真实感。

（3）接着给出的是进行初始化的onSurfaceCreated方法，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_1/src/com.bn.Sample17_1目录下的MySurfaceView.java。

```
1 public void onSurfaceCreated(GL10 gl, EGLConfig config) {
2     ……//此处省略了一些不重要的代码，需要的读者请参考随书光盘
4     floor=new TexFloor(ShaderManager.getTextureShaderProgram(),    //创建地面矩形
4         80*Constant.UNIT_SIZE,-Constant.UNIT_SIZE,planeShape,dynamicsWorld);
6     int size=2;   //立方体箱子尺寸
7     float xStart=(-size/2.0f+0.5f)*(2+0.4f)*Constant.UNIT_SIZE;    //x 坐标起始值
8     float yStart=0.02f;                                        //y 坐标起始值
9     float zStart=(-size/2.0f+0.5f)*(2+0.4f)*Constant.UNIT_SIZE-4f;//z 坐标起始值
10        for(int i=0;i<size;i++){                               //对层、行、列进
11         for(int j=0;j<size;j++){                              //行循环，创建两
12           for(int k=0;k<size;k++){                            //层 8 个立方体箱子
13                TexCube tcTemp=new TexCube(                    //创建立方体箱子
14                        MySurfaceView.this,                    //MySurfaceView 的引用
15                        Constant.UNIT_SIZE,                    //箱子的尺寸
16                        boxShape,                              //碰撞形状
17                        dynamicsWorld,                         //物理世界
18                        1,                                     //刚体质量
19                        xStart+i*(2+0.4f)*Constant.UNIT_SIZE,    //箱子初始 x 坐标
20                        yStart+j*(2.02f)*Constant.UNIT_SIZE,     //箱子初始 y 坐标
21                        zStart+k*(2+0.4f)*Constant.UNIT_SIZE,    //箱子初始 z 坐标
22                         ShaderManager.getTextureShaderProgram()//着色器程序引用
23                );
24                tca.add(tcTemp);                     //将新箱子添加到待添加箱子集合中
25                //将立方体箱子设置为一开始不是激活的
26                tcTemp.body.forceActivationState(RigidBody.WANTS_DEACTIVATION);
27        }}}
28      ……//此处省略了定时进行物理模拟操作的线程，前面已经给出
29   }}
```

说明

上述代码中最重要的就是第10-27行，通过3层嵌套循环创建了两层，8个立方体箱子。另外要特别注意的是，上述代码中在初始状态下将箱子刚体设置为非激活状态。

（4）下面要给出的是触控事件的处理方法onTouchEvent，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_1/src/com.bn.Sample17_1目录下的MySurfaceView.java。

```
public boolean onTouchEvent(MotionEvent e) {
    switch (e.getAction()) {
      case MotionEvent.ACTION_DOWN:                    //处理触控点按下的事件
       TexCube tcTemp=new TexCube(                     //创建一个箱子
           this,                                       //MySurfaceView 的引用
       Constant.UNIT_SIZE,                             //箱子的尺寸
       boxShape,                                       //碰撞形状
       dynamicsWorld,                                  //物理世界
           1,                                          //刚体质量
           0,                                          //箱子初始 x 坐标
           2,                                          //箱子初始 y 坐标
           4,                                          //箱子初始 z 坐标
           ShaderManager.getTextureShaderProgram() //着色器程序引用
       );

           tcTemp.body.setLinearVelocity(new Vector3f(0,2,-12));//设置箱子的初始线速度
           tcTemp.body.setAngularVelocity(new Vector3f(0,0,0)); //设置箱子的初始角速度
           synchronized(tcaForAdd) {                   //锁定待添加箱子集合
             tcaForAdd.add(tcTemp);                    //将新箱子添加到待添加箱子集合中
           }
          break;
      }
      return true;                                     //返回 true 通知系统此事件已被处理
   }
```

说明

此方法中最主要的就是创建了一个新的箱子对象，并设置了新箱子的初速度，最后将新箱子添加进箱子列表中。

17.3.5　水平地面——TexFloor类

了解了实施整体3D场景渲染的MySurfaceView类后，就可以进一步开发组成场景的各个部件 类了。首先是实现水平地面的TexFloor类，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_1/src/com.bn.Sample17_1目录下的TexFloor.java。

```
1 package com.bn.Sample17_1;                                          //声明包
2 ……//此处省略了部分引入相关类的代码，需要的读者请参考随书光盘
3 public class TexFloor {
4 ……// 此处省略了部分成员变量的声明，需要的读者请参考随书光盘
5     public TexFloor(int mProgram,final float UNIT_SIZE,
6         float yOffset,CollisionShape groundShape,DiscreteDynamicsWorld dynamicsWorld) {
7             ……// 此处省略了部分不重要的代码，需要的读者请参考随书光盘
8             Transform groundTransform = new Transform();          //创建刚体的初始变换对象
9             groundTransform.setIdentity();                        //对初始变换对象初始化
10                groundTransform.origin.set(new Vector3f(0.f, yOffset, 0.f));
                                                                  //设置初始的平移变换
11                Vector3f localInertia = new Vector3f(0, 0, 0);   //存储惯性的向量
12                //创建刚体的运动状态对象
13                DefaultMotionState myMotionState = new DefaultMotionState(groundTransform);
14                RigidBodyConstructionInfo rbInfo =               //创建刚体描述信息对象
15                    new RigidBodyConstructionInfo(0, myMotionState, groundShape, localInertia);
16                RigidBody body = new RigidBody(rbInfo);          //创建刚体
17                body.setRestitution(0.4f);                       //设置反弹系数
18                body.setFriction(0.8f);                          //设置摩擦系数
19                dynamicsWorld.addRigidBody(body);                //将刚体添加进物理世界
20                initVertexData(UNIT_SIZE);                       //初始化顶点数据
21                intShader(mProgram);                             //初始化着色器
22        }
23        public void initVertexData(final float UNIT_SIZE){/*代码省略*/} //初始化顶点数据
24        public void initShader(int mProgram) {/*代码省略*/}     //初始化着色器
25        public void drawSelf(int texId) {/*代码省略*/}          //绘制水平地面
26    }
```

- 第8-10行创建并设置了刚体的初始变换对象，其功能为确定刚体初始出现的位置。
- 第16-18行创建了代表水平地面的刚体对象，并设置了其反弹系数与摩擦系数。

> **提示**
>
> 初始化顶点数据的initVertexData方法、初始化着色器的initShader方法以及执行绘制的drawSelf方法与本书前面很多案例中的基本相同，这里不再赘述。

17.3.6 箱子——TexCube类

上一小节介绍了表示水平地面的TexFloor类，本小节将介绍场景中的另一重要部件，表示箱子的TexCube类，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_1/src/com.bn.Sample17_1目录下的TexCube.java。

```
package com.bn.Sample17_1;                               //声明包
……//此处省略了部分引入相关类的代码，需要的读者请参考随书光盘
public class TexCube{
```

```
    TextureRect tr;                                          //用于绘制箱子各个面的纹理矩形
    float halfSize;                                          //立方体箱子的半边长
    RigidBody body;                                          //对应的刚体对象
    int mProgram;                                            //着色器程序引用 id
    MySurfaceView mv;                                        //MySurfaceView 的引用
    public TexCube(MySurfaceView mv,float halfSize,CollisionShape colShape,
      DiscreteDynamicsWorld dynamicsWorld,float mass,float cx,float cy,float cz,int
      mProgram){
      boolean isDynamic = (mass != 0f);                      //物体是否可以运动
      Vector3f localInertia = new Vector3f(0, 0, 0);         //惯性向量
      if (isDynamic){                                        //如果物体可以运动
        colShape.calculateLocalInertia(mass, localInertia);//计算惯性
      }
      Transform startTransform = new Transform();            //创建刚体的初始变换对象
      startTransform.setIdentity();                          //变换初始化
      startTransform.origin.set(new Vector3f(cx, cy, cz));//设置刚体初始的位置
      //创建刚体的运动状态对象
      DefaultMotionState myMotionState = new DefaultMotionState(startTransform);
      RigidBodyConstructionInfo rbInfo = new RigidBodyConstructionInfo
              (mass, myMotionState, colShape, localInertia); //创建刚体描述信息对象
      body = new RigidBody(rbInfo);                          //创建刚体
      body.setRestitution(0.6f);                             //设置反弹系数
      body.setFriction(0.8f);                                //设置摩擦系数
      dynamicsWorld.addRigidBody(body);                      //将刚体添加进物理世界
      this.mv=mv;                                            //保存 MySurfaceView 引用
      tr=new TextureRect(halfSize);                          //创建纹理矩形
      this.mProgram=mProgram;                                //保存着色器程序引用
      this.halfSize=halfSize;                                //保存箱子的半边长
    }
    public void drawSelf(int[] texIda){                      //绘制立方体箱子的方法
      tr.initShader(mv, mProgram);                           //初始化纹理矩形的绘制着色器
      int texId=texIda[0];                                   //箱子运动时的纹理 id
      if(!body.isActive()){texId=texIda[1];}                 //箱子静止时的纹理 id
      MatrixState.pushMatrix();                              //保护现场
      //从物理世界中获取这个箱子对应刚体的变换信息对象
      Transform trans = body.getMotionState().getWorldTransform(new Transform());
     MatrixState.translate(trans.origin.x,trans.origin.y, trans.origin.z); //进行平移变换
      Quat4f ro=trans.getRotation(new Quat4f());             //获取当前旋转变换的信息进入四元数
      if(ro.x!=0||ro.y!=0||ro.z!=0){                         //若四元数 3 个轴的分量不都为 0
          float[] fa=SYSUtil.fromSYStoAXYZ(ro);                 //将四元数转换成 AXYZ 的形式
          MatrixState.rotate(fa[0], fa[1], fa[2], fa[3]); //进行旋转变换
      }
      MatrixState.pushMatrix();                              //保护现场
      MatrixState.translate(0, halfSize, 0);                 //沿 y 轴进行平移
      MatrixState.rotate(-90, 1, 0, 0);                      //绕 x 轴进行旋转
      tr.drawSelf( texId);                                   //绘制箱子的上面
      MatrixState.popMatrix();                               //恢复现场
        ……//此处省略了绘制箱子其他 5 个面的代码，需要的读者请参考随书光盘
      MatrixState.popMatrix();                               //恢复现场
    }
}
```

- 第9-31行为此类的构造器，其主要完成的工作包括两个方面。一个是创建了用于绘制箱子每个面的纹理矩形，另一个是创建了箱子对应的刚体，并进行了相应的初始化，最后将刚体添加进物理世界以备模拟时使用。
- 第32-52行为绘制箱子的drawSelf方法，其中首先从箱子对应的刚体中获取箱子当前的位置与姿态信息，然后根据获取的信息对坐标系进行变换，最后基于变换后的坐标系绘制了箱子。

提示

与本书前面的某些案例相同，立方体箱子虽然看起来是一个整体，但其实是由一个纹理矩形变换到6个面的位置分别绘制的。

17.4　复合碰撞形状的使用

上一节中给出了一个箱子互相碰撞的案例，通过其读者应该掌握了JBullet的基本用法。但箱子互相碰撞的案例中运动的刚体都是单一的立方体形状，这在实际开发中是远远不够的。因此，本节将通过一个案例Sample17_2来介绍复合碰撞形状的使用方法。

17.4.1　案例运行效果

顾名思义，复合碰撞形状提供的就是将多个简单碰撞形状组合成一个整体的能力。本节给出的案例中采用的是将两个圆柱与一个立方体组合而成的复合碰撞形状，其运行效果如图17-20、图17-21、图17-22及图17-23所示。

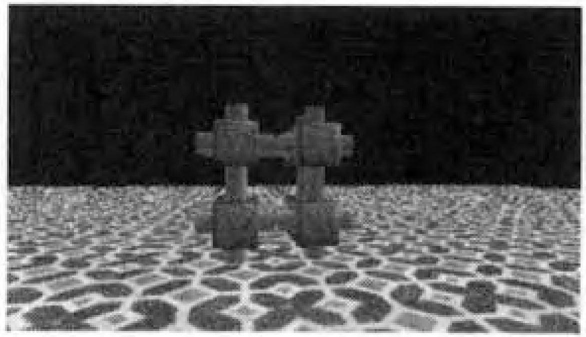


图17-20 案例开始运行

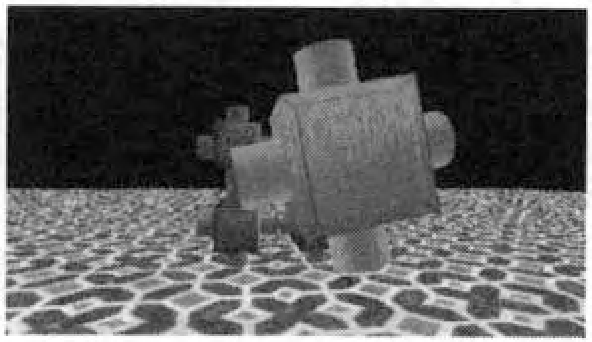


图17-21 抛出撞击复合体

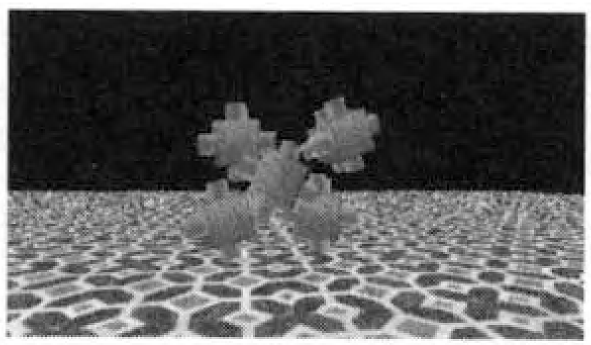


图17-22 开始碰撞

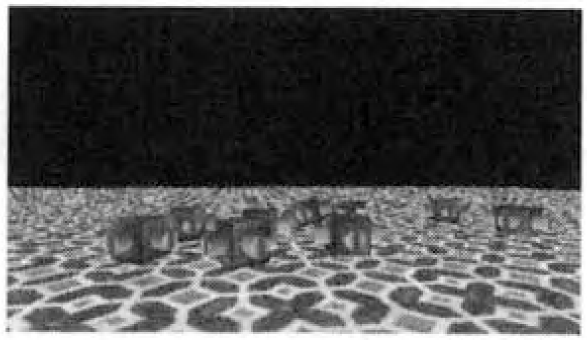


图17-23　碰撞结束

图17-20为案例开始运行时的初始情况，图17-21为手指触控屏幕后抛出一个新的复合形状的情况，图17-22为复合形状相互碰撞时的情况，图17-23为碰撞结束后多个复合形状散落在水平地面上的情况。

> **提示**
>
> 为了便于观察，本案例中对静止的和运动的复合形状采用了不同颜色的纹理。由于插图采用灰度印刷，因此这一点可能看得不是很清楚，此时请读者自行采用真机运行本案例进行观察即可。

17.4.2　立方体圆柱复合形状——CubeCylinder类

从上一小节给出的4幅案例运行效果图中可以看出，本节案例与上一节的案例基本功能非常类 似。主要的区别是上一节案例中抛出的是立方体的箱子，而本节案例中抛出的是由圆柱与立方体组成的复合形状。因此本节案例中的很多代码与上一节案例中的基本相同，对于这些代码这里不再赘述，需要的读者请参考随书光盘。

另外，用于绘制圆柱的的基本形状在前面的章节中也已经介绍过。因此这里需要重点介绍的就是实现立方体圆柱复合形状的CubeCylinder类，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_2/src/com.bn.Sample17_2目录下的CubeCylinder.java。

```java
package com.bn.Sample17_2;                                        //声明包
……//此处省略了部分引入相关类的代码，需要的读者请参考随书光盘
public class CubeCylinder{
  Cylinder cl;                                                   //用于绘制的圆柱
  TexCube tc;                                                    //用于绘制的立方体箱子
  float halfSize;                                                //半边长
  RigidBody body;                                                //对应的刚体对象
  MySurfaceView mv;                                              //MySurfaceView 的引用
  public CubeCylinder(MySurfaceView mv,float halfSize,CollisionShape[] csa,
    DiscreteDynamicsWorld dynamicsWorld,float mass,float cx,float cy,float cz,int[]
    mProgram){
    CompoundShape comShape=new CompoundShape();                  //创建组合形状对象
    Transform localTransform = new Transform();                  //创建变换对象
    localTransform.setIdentity();                                //初始化变换
    localTransform.origin.set(new Vector3f(0, 0, 0));            //设置变换的平移参数
    comShape.addChildShape(localTransform, csa[0]);              //添加子形状(箱子)
    comShape.addChildShape(localTransform, csa[1]);              //添加子形状(围绕 X 轴的圆柱)
    localTransform = new Transform();                            //创建变换对象
    localTransform.basis.rotX((float)Math.toRadians(90));        //绕 x 旋转 90°
    comShape.addChildShape(localTransform, csa[2]);              //添加子形状(围绕 Y 轴的圆柱)
    boolean isDynamic = (mass != 0f);                            //判断刚体是否可运动
    Vector3f localInertia = new Vector3f(0, 0, 0);               //创建存放惯性的向量
    if (isDynamic){                                              //如果刚体可以运动
      comShape.calculateLocalInertia(mass, localInertia);        //计算刚体的惯性
    }
    Transform startTransform = new Transform();                  //创建刚体的初始变换对象
    startTransform.setIdentity();                                //初始化变换对象
    startTransform.origin.set(new Vector3f(cx, cy, cz));         //设置刚体的初始位置
    //创建刚体的运动状态对象
    DefaultMotionState myMotionState = new DefaultMotionState(startTransform);
    RigidBodyConstructionInfo rbInfo =                           //创建刚体描述信息对象
        new RigidBodyConstructionInfo (mass, myMotionState, comShape, localInertia);
    body = new RigidBody(rbInfo);                                //创建刚体对象
    body.setRestitution(0.6f);                                   //设置反弹系数
    body.setFriction(0.8f);                                      //设置摩擦系数
    dynamicsWorld.addRigidBody(body);                            //将刚体添加进物理世界
    tc=new TexCube(halfSize,mProgram[0]);                        //创建绘制用立方体箱子
    cl=new Cylinder(halfSize/2,halfSize*3.6f,16,mProgram[0],mv);//创建绘制用圆柱
    this.halfSize=halfSize;                                      //保存半边长
```

```
39    this.mv=mv;                                              //保存 MySurfaceView 的引用
40  }
41  public void drawSelf(int[] texIda,int[] texIdb){           //绘制复合形状的方法
42    int texId1=texIda[0];                                    //立方体箱子运动时的纹理
43    int texId2=texIdb[1];                                    //圆柱运动时的纹理
44    if(!body.isActive()){                                    //若刚体静止
45      texId1=texIda[1];                                      //立方体静止时的纹理
46      texId2=texIdb[0];                                      //圆柱静止时的纹理
47    }
48    MatrixState.pushMatrix();                                //保护现场
49    //获取这个物体对应刚体的变换信息对象
50    Transform trans=body.getMotionState().getWorldTransform(new Transform());
51   MatrixState.translate(trans.origin.x,trans.origin.y, trans.origin.z); //进行平移变换
52    Quat4f ro=trans.getRotation(new Quat4f());          //获取当前旋转变换的信息进入四元数
53    if(ro.x!=0||ro.y!=0||ro.z!=0){                      //若四元数 3 个轴的分量不都为 0
54      float[] fa=SYSUtil.fromSYStoAXYZ(ro);             //将四元数转换成 AXYZ 的形式
55      MatrixState.rotate(fa[0], fa[1], fa[2], fa[3]);    //进行旋转变换
56    }
57    tc.drawSelf( texId1);                                    //绘制立方体箱子
58    MatrixState.pushMatrix();                                //保护现场
69    MatrixState.rotate(90,0, 0, 1);                          //绕 Z 轴旋转 90°
60    MatrixState.translate(0, -halfSize*1.8f, 0);             //沿 Y 轴进行平移
61    cl.drawSelf(texId2,  texId2,  texId2);                   //绘制绕 X 轴圆柱
62    MatrixState.popMatrix();                                 //恢复现场
63    MatrixState.pushMatrix();                                //保护现场
64    MatrixState.translate(0, -halfSize*1.8f, 0);             //沿 Y 轴进行平移
65    cl.drawSelf(texId2,  texId2,  texId2);                   //绘制绕 Y 轴的圆柱
66    MatrixState.popMatrix();                                 //恢复现场
67    MatrixState.popMatrix();                                 //恢复现场
68  }
69 }
```

- 第9-40行为该类的构造器，其中最重要的工作就是将两根圆柱与一个立方体组合而形成了本案例中所需要的复合形状。得到复合形状后又基于其创建了刚体，然后对刚体进行了初始化设置，最后将刚体添加进了物理世界。
- 第41-68行为绘制复合形状的drawSelf方法，其中首先从复合形状对应的刚体中获取复合形状当前的位置与姿态信息，然后根据获取的信息对坐标系进行变换，最后基于变换后的坐标系绘制了组成复合形状的立方体与圆柱。

提示

要注意的是，本案例对于复合形状中的圆柱绘制采用的是圆柱，碰撞检测采用的也是圆柱。若希望提高一些性能则可以将用于

圆柱碰撞检测的形状改为胶囊，有兴趣的读者可以动手尝试，只需要修改MySurfaceView.java中创建基本碰撞形状的代码即可。

17.5 凹凸地形的案例

前面两节案例的场景中，采用的都是水平的地面。但实际开发中地面不一定是水平的，可能会是凹凸不平的丘陵。本节将介绍一个通过使用BvhTriangleMeshShape类实现凹凸地形碰撞的案例Sample17_3，掌握了此案例中介绍的知识后，读者模拟现实世界的能力将进一步加强。

17.5.1 案例运行效果

介绍本案例的具体开发之前，需要首先了解一下本节案例的运行效果，如图17-24、图17-25、图17-26和图17-27这4幅图所示。

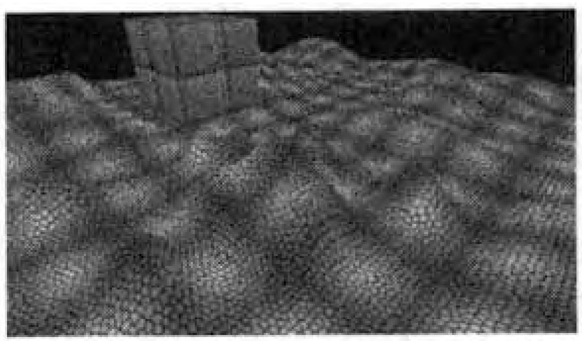


图17-24　案例开始运行

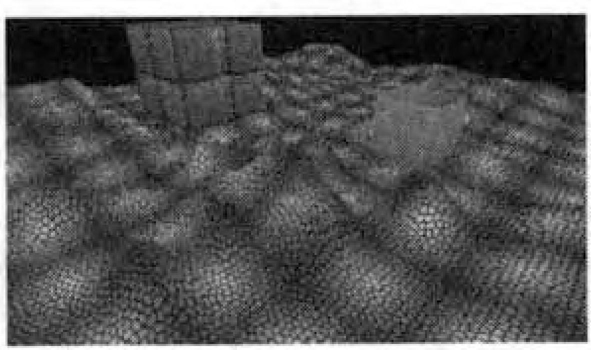


图17-25　抛出撞击箱子

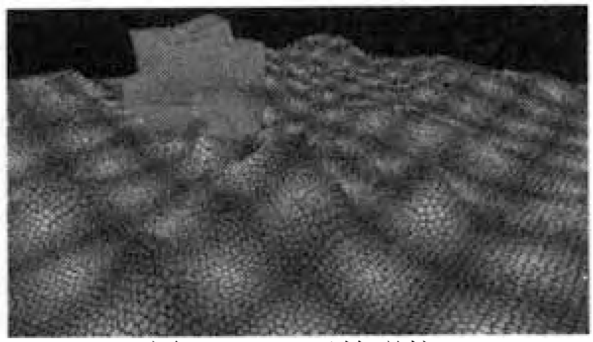


图17-26　开始碰撞

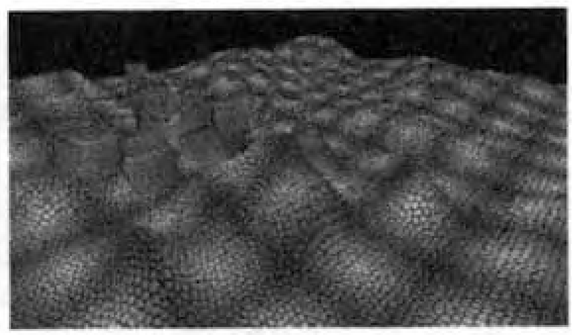


图17-27　碰撞结束

图17-24为案例开始运行时的初始情况，图17-25为手指触控屏幕后抛出一个新箱子的情况，图17-26为箱子、凹凸地面相互碰撞时的情况，图17-27为碰撞结束后多个箱子散落在凹凸不平的地面上的情况。

提示

为了便于观察，本案例中对静止的和运动的复合形状采用了不同颜色的纹理。由于插图采用灰度印刷，因此这一点可能看得不是很清楚，此时请读者自行采用真机运行本案例进行观察即可。

17.5.2　地形类——LandForm

从本节案例的4幅运行效果图中可以看出，本节案例的基本功能与前面17.3节案例的相同，只是将17.3节案例中的水平地面替换成了凹凸不平的地面。因此，这里仅给出实现凹凸不平地面的LandForm类，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_3/src/com.bn.Sample17_3目录下的LandForm.java。

```
package com.bn.Sample17_3;                          //声明包
……//此处省略了部分引入相关类的代码，需要的读者请参考随书光盘
public class LandForm {
  ……//此处省略了部分声明成员变量的代码，需要的读者请参考随书光盘
  public LandForm(MySurfaceView mv,final float UNIT_SIZE, float yOffset,
      DiscreteDynamicsWorld dynamicsWorld,  float[][] yArray,int rows,int cols,int
mProgram){
    this.mProgram=mProgram;                         //保存着色器程序引用
    ……//此处省略了基于yArray生成地形顶点坐标的代码，需要的读者请参考随书光盘
    ByteBuffer gVertices=                           //创建为凹凸地形碰撞形状服务的顶点数据缓冲
```

```
10              ByteBuffer.allocateDirect(vCount*3*4).order(ByteOrder.nativeOrder());
11      for(int i=0;i<vertices.length;i++){ gVertices.putFloat(i*4,vertices[i]); }
                                                            //将顶点数据添加到缓冲中
12      gVertices.position(0);                              //设置缓冲区起始位置
13      ByteBuffer gIndices=            //创建为凹凸地形碰撞形状服务的顶点索引缓冲
14              ByteBuffer.allocateDirect(vCount*4).order(ByteOrder.nativeOrder());
15      for(int i=0;i<vCount;i++){ gIndices.putInt(i); }    //将索引数据添加到缓冲中
16      gIndices.position(0);                               //设置缓冲区起始位置
17      int vertStride = 4*3;                               //顶点数据间隔
18      int indexStride = 4*3;                              //索引数据间隔
19      TriangleIndexVertexArray indexVertexArrays=         //创建三角形组顶点数据数组
20        new TriangleIndexVertexArray(
21         vCount/3,                                        //三角形的个数
22         gIndices,                                        //索引缓冲
23         indexStride,                                     //索引数据间隔
24         vCount,                                          //顶点个数
25         gVertices,                                       //顶点缓冲
26         vertStride                                       //顶点数据间隔
27       );
28      //创建凹凸地形对应的碰撞形状
29     CollisionShape groundShape=new BvhTriangleMeshShape(indexVertexArrays,true,true);
30      Transform groundTransform = new Transform();        //创建刚体的初始变换对象
31      groundTransform.setIdentity();                      //初始化变换
32     groundTransform.origin.set(new Vector3f(0.f, 0.f, 0.f));//设置刚体(凹凸地形)的初始位置
33      Vector3f localInertia = new Vector3f(0, 0, 0);      //创建存放惯性的向量
34      //创建刚体的运动状态对象
35      DefaultMotionState myMotionState = new DefaultMotionState(groundTransform);
36      RigidBodyConstructionInfo rbInfo =                  //创建刚体描述信息对象
37              new RigidBodyConstructionInfo (0, myMotionState, groundShape, localInertia);
38      RigidBody body = new RigidBody(rbInfo);             //创建刚体
39      body.setRestitution(0.4f);                          //设置反弹系数
40      body.setFriction(0.8f);                             //设置摩擦系数
41     body.setCollisionFlags(body.getCollisionFlags() & ~CollisionFlags.KINEMATIC_OBJECT);
42      body.forceActivationState(CollisionObject.ACTIVE_TAG);
43      dynamicsWorld.addRigidBody(body);                   //将刚体添加进物理世界
44      initShader (mv);                                    //初始化着色器程序
45   }
46   ……//此处省略了部分方法，需要的读者请参考随书光盘
47 }
```

- 第8行省略了基于yArray生成地形顶点坐标的代码，这部分代码与前面第11章中灰度图地形案例中的一致。同时yArray也是基于灰度图地形技术从灰度图中加载的，需要的读者可以参考第11章中灰度图地形的相关部分或随书光盘中的源代码。
- 第9-16行为为凹凸地形碰撞形状准备顶点坐标数据缓冲与顶点索引数据缓冲的代码。
- 第19-27行基于前面准备的两个数据缓冲创建了三角形组顶点数据数组，为后面创建凹凸地形碰撞形状做准备。
- 第29-43行首先基于三角形组顶点数据数组创建了凹凸地形碰撞形状，然后基于其创建了刚 体，接着对刚体进行了初始化设置，最

后将刚体添加进了物理世界。

17.6　任意形状物体的碰撞

上一节介绍了使用BvhTriangleMeshShape类实现凹凸地形，其实使用BvhTriangleMeshShape类不但可以实现凹凸地形，还可以实现任意的形状的静态物体。但实际开发中有时还可能有开发任意形状运动物体的需求，此时就需要使用GlmpactMeshShape类了，本节将给出一个具体的案例Sample 17_4。

17.6.1　案例运行效果

介绍本节案例的具体开发之前，需要首先了解一下其运行效果，如图17-28、图17-29、图17-30和图17-31这4幅图所示。

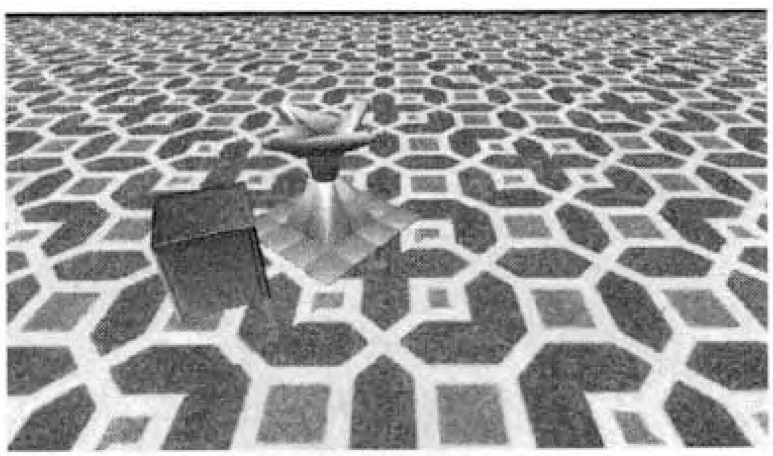
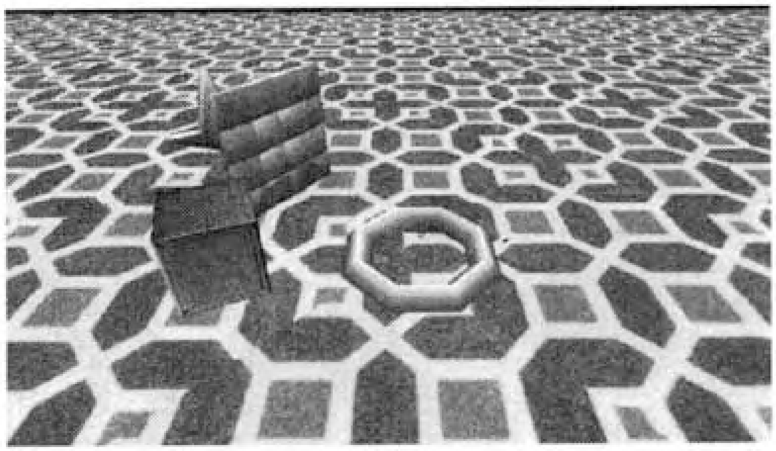
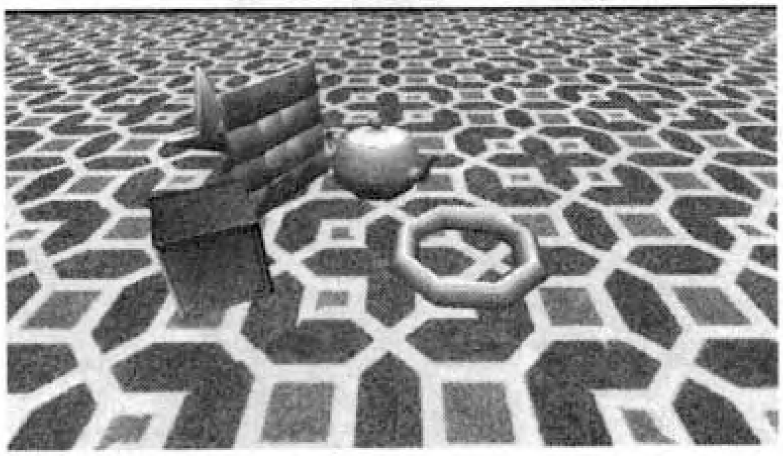
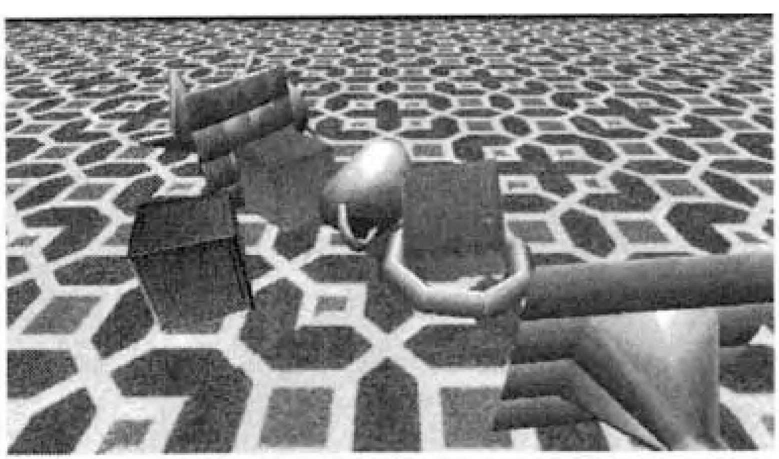


图17-28　抛出箱子、桌子后的效果图


图17-29 再抛出圆环后的效果图


图17-30 再抛出茶壶后的效果图


图17-31 抛出更多物体的效果图

提示

任意形状物体的碰撞计算量很大，执行起来可能会比较慢。读者运行案例时不要太着急，用手指触控屏幕抛出物体的频率不要太高。

17.6.2 加载物体类——LoadedObjectVertexNormal

从上一小节的4幅运行效果图中读者应该已经猜到，本节案例中抛出的桌子、茶壶、圆环等任意形状的物体其顶点信息数据是从Obj文件中加载的。另外，从4幅运行效果途中可以看出，本节案例的基本功能与前面17.3节案例是十分类似的，只不过17.3节案例抛出的是立方体箱子，而本节案例抛出的是任意形状的物体。

因此，本节案例中的很多代码与前面的案例相同，故这里仅着重介绍与GlmpactMeshShape类 使用密切相关的部分代码。首先需要介绍的是从Obj文件中加载而得到的任意形状物体对象对应的LoadedObjectVertexNormal类，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_4/src/com.bn.Sample17_4目录下的LoadedObjectVertexNormal.java。

```
1package com.bn.Sample17_4;
2……//此处省略了部分引入相关类的代码，需要的读者请参考随书光盘
3public class LoadedObjectVertexNormal {
4    ……//此处省略了部分成员变量的声明，需要的读者请参考随书光盘
5    int vCount=0;                                          //顶点数量
6    CollisionShape loadShape;                              //碰撞形状引用
7    float[] vertices;                                      //顶点坐标数据数组
8    float[] normals;                                       //顶点法向量数据数组
9    public LoadedObjectVertexNormal(float[] vertices,float[] normals)  {
10          ……//此处省略了对顶点坐标缓冲和顶点法向量缓冲初始化的代码
11          //创建为任意碰撞形状(GImpact 三角网状形状)服务的顶点数据缓冲
12          ByteBuffer gVertices=ByteBuffer.allocateDirect(vCount*3*4).order(ByteOrder.
            nativeOrder());
13          for(int i=0;i<vertices.length;i++){
15            gVertices.putFloat(i*4,vertices[i]);         //将顶点数据添加到缓冲中
16          }
17          gVertices.position(0);                         //设置缓冲区起始位置
18          //创建为任意碰撞形状(GImpact 三角网状形状)服务的顶点索引缓冲
19          ByteBuffer gIndices=ByteBuffer.allocateDirect(vCount*4).order(ByteOrder.
            nativeOrder());
20          for(int i=0;i<vCount;i++){                     //将索引数据添加到缓冲中
22              gIndices.putInt(i);
23          }
24          gIndices.position(0);                          //设置缓冲区起始位置
25          int vertStride = 4*3;                          //顶点数据间隔
26              int indexStride = 4*3;                     //索引数据间隔
27          TriangleIndexVertexArray indexVertexArrays=    //创建三角形组顶点数据数组
28              new TriangleIndexVertexArray(
29                  vCount/3,                              //三角形的个数
30                  gIndices,                              //索引缓冲
31                  indexStride,                           //索引数据间隔
32                  vCount,                                //顶点个数
33                  gVertices,                             //顶点缓冲
34                  vertStride                             //顶点数据间隔
35              );
36          //加载的任意形状的物体对应的碰撞形状
37          GImpactMeshShape trimesh = new GImpactMeshShape(indexVertexArrays);
38          trimesh.updateBound();                         //更新碰撞形状边界
39          loadShape =trimesh;                            //保存碰撞形状引用
40        }
41        public void intShader(int mProgram) {/*代码省略*/} //初始化着色器的方法
42        public void drawSelf() {/*代码省略*/}            //绘制加载物体的方法
43    }
```

说明

上述代码中最重要的就是第11-39行的代码，其中首先将顶点的数据放到了顶点数据缓冲与顶点索引缓冲中，然后基于两个缓冲创建了三角形组顶点数据数组，最后基于三角形组顶点数据数组创建了对应到加载物体的有三角形组确定的碰撞形状。

17.6.3 加载物体刚体类——LoadRigidBody

上一小节介绍的LoadedObjectVertexNormal类完成了任意形状物体的绘制和对应碰撞形状对象的生成，本小节将介绍加载模型对应的刚体类LoadRigidBody，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_6/src/com.bn.Sample17_6目录下的LoadRigidBody.java。

```
package com.bn.Sample17_4;                                     //声明包
……//此处省略了部分引入相关类的代码，需要的读者请参考随书光盘
public class LoadRigidBody{
  LoadedObjectVertexNormal lovo;                               //加载物体对象引用
  RigidBody body;                                              //对应的刚体对象引用
  int mProgram;                                                //着色器程序引用
  public LoadRigidBody(int mProgram,float mass, LoadedObjectVertexNormal lovo,
    float cx,float cy,float cz,DiscreteDynamicsWorld dynamicsWorld){
    this.lovo=lovo;                                            //保存加载物体对象引用
    this.mProgram=mProgram;                                    //保存着色器程序引用
    CollisionShape colShape=lovo.loadShape;                    //保存碰撞形状引用
    boolean isDynamic = (mass != 0f);                          //确定物体是否可运动
    Vector3f localInertia = new Vector3f(0f, 0f, 0f);          //创建存放惯性的向量
    if (isDynamic) {colShape.calculateLocalInertia(mass, localInertia);}//计算惯性
    Transform startTransform = new Transform();                //创建刚体的初始变换对象
    startTransform.setIdentity();                              //初始化变换对象
    startTransform.origin.set(new Vector3f(cx, cy, cz));       //设置刚体的初始位置
    //创建刚体的运动状态对象
    DefaultMotionState myMotionState = new DefaultMotionState(startTransform);
    RigidBodyConstructionInfo cInfo =                          //创建刚体描述信息对象
              new RigidBodyConstructionInfo (mass, myMotionState, colShape, localInertia);
    body = new RigidBody(cInfo);                               //创建刚体对象
    body.setRestitution(0.4f);                                 //设置反弹系数
    body.setFriction(0.8f);                                    //设置摩擦系数
    dynamicsWorld.addRigidBody(body);                          //将刚体添加进物理世界
  }
  public void drawSelf(){                                      //绘制刚体
    lovo.initShader(mProgram);                                 //初始化着色器
    MatrixState.pushMatrix();                                  //保护现场
    //获取此刚体对应的变换信息对象
    Transform trans = body.getMotionState().getWorldTransform(new Transform());
   MatrixState.translate(trans.origin.x,trans.origin.y, trans.origin.z); //进行平移变换
    Quat4f ro=trans.getRotation(new Quat4f());        //获取当前旋转变换的信息进入四元数
    if(ro.x!=0||ro.y!=0||ro.z!=0){                    //若四元数3个轴的分量不都为0
      float[] fa=SYSUtil.fromSYStoAXYZ(ro);           //将四元数转换成AXYZ形式
```

```
36        MatrixState.rotate(fa[0], fa[1], fa[2], fa[3]);     //进行旋转变换
37      }
38      lovo.drawSelf();                                     //绘制加载的物体
39      MatrixState.popMatrix();                             //恢复现场
40    }
41 }
```

- 第12-25行基于从构造器参数接收的碰撞形状创建了对应的刚体对象，并对刚体对象进行了必要的初始化设置，最后将刚体对象添加进了物理世界。
- 第27-39行为根据物理模拟的情况绘制加载物体的drawSelf方法，其中首先从物体对应的刚体中获取物体当前的位置与姿态信息，然后根据获取的信息对坐标系进行变换，最后基于变换后的坐标系绘制了物体。

提示

本案例中加载的物体有3种外观，包括茶壶、圆环与桌子，分别对应到项目assets目录下的ch.obj、yh.obj、table.obj3个模型文件。但本案例的能力并不仅限于这3种外观，有兴趣的读者可以用3DMax等建模工具设计其他任何外观的物体，并导出为obj文件。再将obj文件导入项目，对源代码稍作修改即可让此应用程序抛出自己所期望的任何形状的物体了。

17.7　铰链关节

前面几节给出的案例中都是独立的刚体之间的相互碰撞，刚体与刚体之间除了碰撞之外没有其他逻辑联系。但现实世界中不仅如此，还有很多情况下需要把多个刚体按照某种规则联系在一起，如门和门框、自行车的车轮和车身、机械手臂中的各个组成部件等。

想模拟这些机械结构就需要用到JBullet中提供的关节，JBullet中提供的关节有很多种，主要包括：铰链关节、滑动关节、六自由度关节、点对点关节等。本节将通过一个案例Sample17_5向读者介绍铰链关节的具体使用方法。

17.7.1　铰链关节的基本知识

介绍具体的案例之前，首先应该了解一下铰链关节的基本知识。铰链关节是仅有一个旋转自由度的关节，通过铰链约束限制相关刚体仅能绕铰链轴旋转，其原理如图17-32所示。

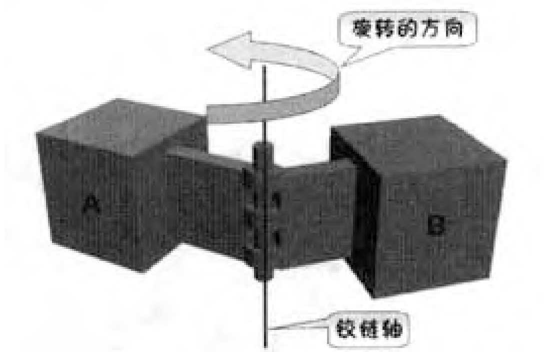


图17-32　铰链关节原理图

从图17-32中可以看出，日常生活中有不少需要用到铰链关节的情况，如门和门框之间就是如此。

> **提示**
>
> 从某种角度来说，关节可以看作刚体之间的约束。因此，JBullet中的各种关节都是通过在相关刚体间添加相应约束来实现的。

实现铰链关节需要用到的约束为HingeConstraint，其构造器及常用方法如表17-15所列。

表17-15　HingeConstraint类的构造器和常用方法

构造器或方法签名	含　义	类型
public HingeConstraint (RigidBody rbA,	创建添加到一个刚体上的铰链约束。参数rbA代表要添加约束的刚体，参数	构造器

Transform rbAFrame)	rbAFrame代表从约束位置到刚体质心位置的变换	
public HingeConstraint (RigidBody rbA,RigidBody rbB, Transform rbAFrame, Transform rbBFrame)	创建两个刚体之间的铰链约束。参数rbA代表第一个要添加约束的刚体，参数rbB代表第二个要添加约束的刚体，参数rbAFrame代表从约束位置到第一个刚体质心位置的变换，参数rbBFrame代表从约束位置到第二个刚体质心位置的变换	构造器
public float getHingeAngle()	获取铰链关节当前的旋转角度值	方法
public void setLimit (float low, float high)	设置铰链关节的转动范围。参数low代表转动角度的下限值，参数high代表转动角度的上限值	方法
public float getLowerLimit()	获取转动角度的下限值	方法
public float getUpperLimit()	获取转动角度的上限值	方法
public Transform getAFrame (Transform out)	获取从约束位置到第一个刚体质心位置的变换，参数out为获取的变换信息所在的变换对象	方法
public Transform getBFrame (Transform out)	获取从约束位置到第二个刚体质心位置的变换，参数out为获取的变换信息所在的变换对象	方法
public void enableAngularMotor (boolean enableMotor, float targetVelocity, float maxMotorImpulse)	启动马达的方法。参数enableMotor为是否允许关节使用马达的标志，参数targetVelocity为关节旋转的角速度，参数maxMotorImpulse为关节马达的驱动力值	方法

提示

马达是一个非常形象的概念，用于模拟现实世界中用于提供动力的部件，如电动机、柴油机等。几乎每种物理引擎中都有马达的实现，用于根据设定值及关节的类型驱动与关节相关的刚体运动。

17.7.2 案例的运行效果

介绍案例的具体开发之前，应该首先了解一下本节案例的运行效果，具体情况如图17-33与图17-34这两幅图所示。

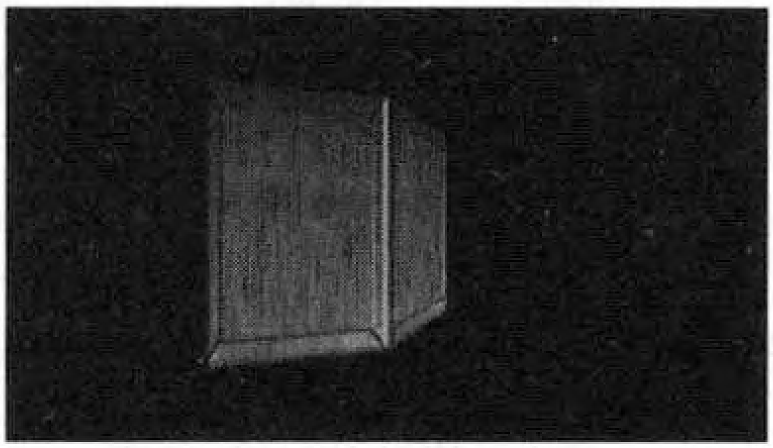


图17-33　模拟门的转动图（1）

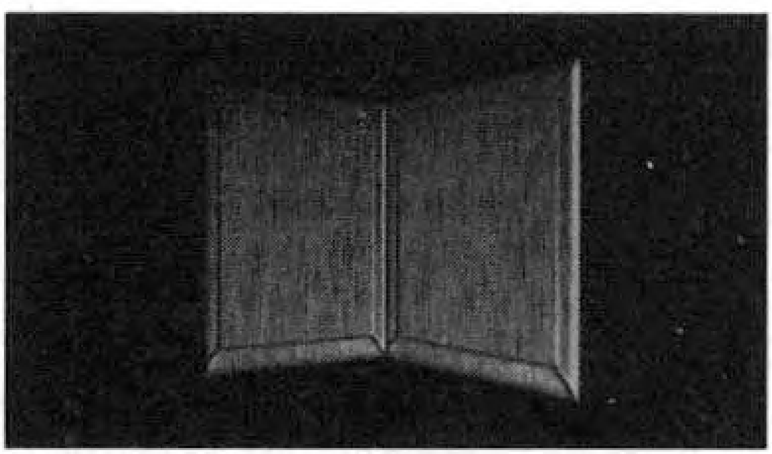


图17-34　模拟门的转动图（2）

说明

从这两幅图中可以看出，本节案例通过铰链关节实现了类似于门开关的一个场景。掌握了本案例后，读者还可以利用铰链关节模拟其他现实世界中的相关场景。另外，案例运行时用手指在屏幕上水平滑动摄像机会绕场景水平旋转，用手指在屏幕上竖直滑动摄像机会上升或下降。

17.7.3　铰链关节旋转角速度的计算

从上一小节给出的运行效果图中可以看出，本案例中右侧的门会绕铰链轴作周期性的往复旋转运动。可以将门的旋转分为1、2、3、4 4个子步骤，如图17-35所示。

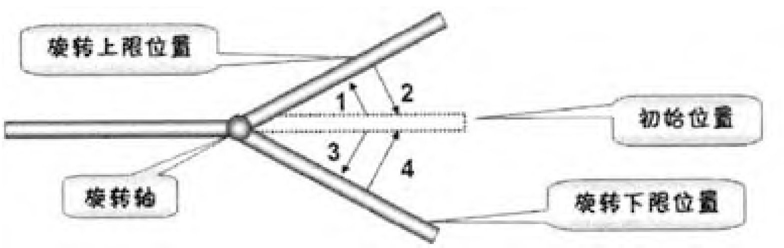


图17-35　门旋转4个子步骤的俯视示意图

从图17-35中可以看出，子步骤1中右侧的门从初始位置绕轴旋转到上限位置，子步骤2中右侧的门绕轴从上限位置旋转回初始位置，子步骤3中右侧的门从初始位置绕轴旋转到下限位置，子步骤4中右侧的门从旋转下限位置绕轴旋转回初始位置。

驱动与铰链关节相关的刚体转动最主要的就是给出当前关节转动的角速度，然后通过enableAngularMotor方法传递给关节。了解了上述内容后，读者可能会很直观地想到：对1、2、3、4 4个阶段分别设置恒定的角速度不就可以达到目的了吗？这样当然是可以的，但模拟的效果就会显得比较假，因为这种情况下角速度是不连续的，如图17-36所示。

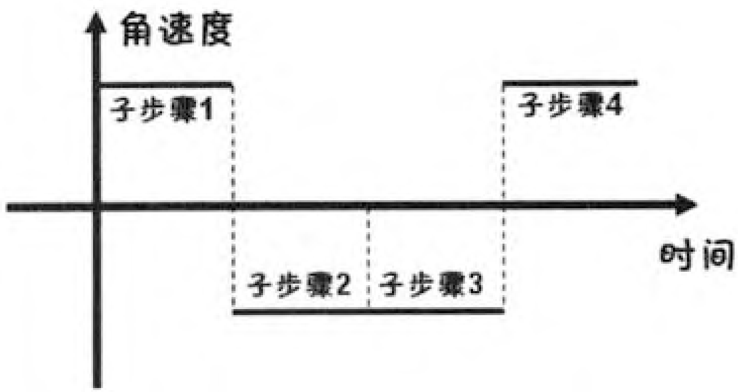


图17-36　直观的角速度设定策略

从图17-36中可以看出，这种策略并不符合现实世界的物理规律，现实世界中的角速度、速度等物理量是连续变化的，不可能有无穷大的角加速度或加速度使得角速度、速度跳跃变化。因此若想模拟的效果比较真实，就应该给出一套角速度连续变化的方案。例如，让旋转角度随时间的变化符合正弦三角函数，如图17-37所示。那么其一阶导数（也就是角速度）的变化即符合余弦三角函数，是连续变化的。

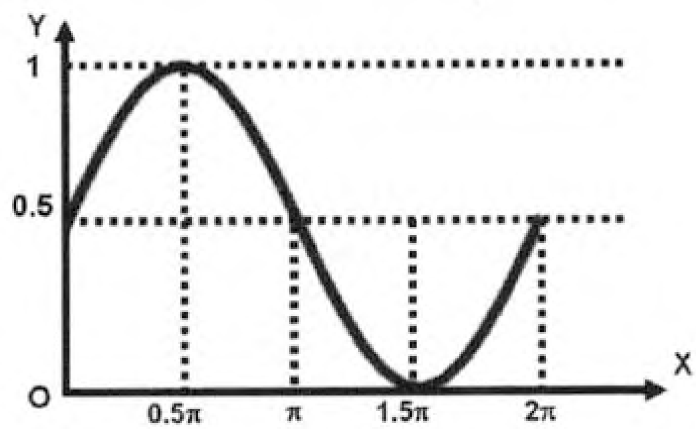


图17-37　按照正弦三角函数规律变化的角度

图17-37中X 轴的值表示当前时刻在周期中的折算值，Y 轴的值表示此时刻对应的旋转角度折算值，x与y之间的关系可以用如下公式来描述。

$$y = 0.5(1+sin(x))$$ 【公式1】

要注意的是，利用此公式计算y时要首先需要将x值折算到0～2π的范围内，计算出y后还需要进行折算才能得到当前时刻对应的旋转角度，折算公式如下。

当前时刻的旋转角度=下限值+*y* ×（上限值-下限值）　　【公式2】

综上所述，可以总结出计算某时刻角速度的步骤如下。

- 给出门运动周期的时长，使用一个计时器统计门转动的时间。接着将门转动的时间对周期求模，然后对周期做除法并乘以2π，即可得到x值。
- 将x值代入公式1即可求得y值，再将y值带入【公式2】即可求出当前时刻门的旋转角度。
- 将当前时刻的旋转角度与上一时刻（要求上一时刻与此时刻的时间差极小）的旋转角度求 差，并将差值除以两个时刻的时间差即可得到当前所需的角速度。

> **提示**
>
> 上述步骤中的最后一步实际上是实现了对旋转角度基于时间求微分（或称之为求导数），对于微分或导数不太熟悉的读者请参考相关的数学资料。另外要注意的是，对速度、角速度等物理量进行模拟时至少要做到一阶连续，否则效果会很假。

17.7.4　3D界面渲染类——MySurfaceView

通过前面的几个小节的介绍，读者应该对铰链关节有了一定的了解，下面将进一步介绍本节案例的开发。由于本节案例中很多类的代码与前面很多案例中的非常相似，因此这里重点介绍有代表性的使用铰链关节的MySurfaceView类，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_7/src/com.bn.Sample17_5目录下的MySurface View.java。

```java
package com.bn.Sample17_5;
……//此处省略了部分引入相关类的代码，需要的读者请参考随书光盘
public class MySurfaceView extends GLSurfaceView {
    DiscreteDynamicsWorld dynamicsWorld;                    //物理世界
    CollisionShape boxShape;                                //长方体碰撞形状
    HingeConstraint constraint;                             //铰链关节约束
    private float time;                                     //当前时间
    private float cyclePeriod;                              //周期时长
    private float muscleStrength;                           //驱动关节的力度
        boolean flag=true;                                  //循环控制标志
        Clock clock = new Clock();                          //时钟（JBullet 提供）
        ……//此处省略了几个不重要的成员变量的声明，需要的读者请参考随书光盘
        public MySurfaceView(Context context) {
            ……//此处省略了一些与前面案例相同的代码，需要的读者请参考随书光盘
            initWorld();                                    //初始化物理世界
            time = 0.0f;                                    //将当前时间初始化为 0
            cyclePeriod = 3000.0f;                          //设置周期时长
            muscleStrength = 5f;                            //设置驱动关节的力度
        }
        @Override
        public boolean onTouchEvent(MotionEvent e) {/*代码省略*/}//触控事件处理方法
        public void initWorld(){/*代码省略*/}//初始化物理世界的方法，与前面案例中的基本相同
        private class MyRenderer implements GLSurfaceView.Renderer{
          Cuboid door1;                                     //表示不动的门的纹理长方体
          Cuboid door2;                                     //表示运动的门的纹理长方体
          int activeTexId;                                  //门的纹理 id
          @Override
          public void onDrawFrame(GL10 gl) {                //绘制一帧画面的方法
             GLES20.glClear(GLES20.GL_DEPTH_BUFFER_BIT|GLES20.GL_COLOR_BUFFER_BIT);
             float ms = getDeltaTimeMicroseconds();         //获取两帧画面间的时间间隔
             time+=ms;                                      //当前时间递增
             float preAngle = constraint.getHingeAngle();      //获取关节上一时刻的角度
```

```
            //求得公式 1 中需要的 x
            float x=(((int)(time / 1000) % (int)(cyclePeriod)) / cyclePeriod)*
            BulletGlobals.SIMD_2_PI;
            float y= 0.5f * (1.0f + (float)Math.sin(x));     //根据公式 1 求出 y
            float currAngle = constraint.getLowerLimit() +  //计算出当前时刻的旋转角度
                y * (constraint.getUpperLimit() - constraint.getLowerLimit());
            float angleDelta = currAngle - preAngle;         //计算角度差
            float desiredAngularVel = 1000000.f * angleDelta/ms;//计算此时的角速度
            constraint.enableAngularMotor(true, desiredAngularVel, muscleStrength);
                                                             //设置旋转马达
           MatrixState.setCamera(cx,cy,cz,0,0,0,0f,1.0f,0.0f);  //设置摄像机
            MatrixState.pushMatrix();                        //保护现场
            door1.drawSelf(activeTexId);                     //绘制不动的门
            door2.drawSelf(activeTexId);                     //绘制运动的门
            MatrixState.popMatrix();                         //恢复现场
        }
        @Override
        public void onSurfaceChanged(GL10 gl, int width, int height) {/*代码省略*/}
        @Override
        public void onSurfaceCreated(GL10 gl, EGLConfig config){
            ……//此处省略了一些初始化的代码，与前面很多案例中的相同
          door1 = new Cuboid(dynamicsWorld,              //创建不动的门对应的长方体
              boxShape, 0, 0, 0, 4, MySurfaceView.this,halfX,halfY,halfZ);
          door2 = new Cuboid(dynamicsWorld,              //创建运动的门对应的长方体
              boxShape, 1, 0, 0, -4, MySurfaceView.this,halfX,halfY,halfZ);
          addHingeConstraint(door1.body,door2.body);  //对两个门对应的刚体添加铰链约束
          ……//此处省略了定时进行物理模拟操作的线程，与前面案例中的相同
        }}
        public void addHingeConstraint(RigidBody rbA,RigidBody rbB){//添加铰链约束的方法
            Transform transA = new Transform();      //创建变换对象(从约束到不动门的质心)
            transA.setIdentity();                    //初始化变换对象
            //设置变换对象中的旋转信息
            MatrixUtil.setEulerZYX(transA.basis, BulletGlobals.SIMD_HALF_PI,0,0 );
            transA.origin.set(new Vector3f(0,0,-halfZ));//设置变换对象中的平移信息
            Transform transB = new Transform();      //创建变换对象(从约束到运动门的质心)
            transB.setIdentity();                    //初始化变换对象
            //设置变换对象中的旋转信息
            MatrixUtil.setEulerZYX(transB.basis, BulletGlobals.SIMD_HALF_PI,0,0 );
            transB.origin.set(new Vector3f(0,0,halfZ)); //设置变换对象中的平移信息
            constraint = new HingeConstraint(rbA,rbB,transA,transB); //创建铰链约束
            constraint.setLimit(             //设置铰链约束的转动范围(即下限值与上限值)
                    -BulletGlobals.SIMD_HALF_PI/2, BulletGlobals.SIMD_HALF_PI/2);
            dynamicsWorld.addConstraint(constraint, true);  //将约束添加进物理世界
        }
        public int initTexture(int drawableId){/*代码省略*/} //加载纹理的方法
        public float getDeltaTimeMicroseconds() {   //获取两帧画面间物理世界时间差的方法
            float dt = clock.getTimeMicroseconds();//获取从时钟创建或上次时钟重置到现在的毫秒数
            clock.reset();                           //重置时钟
            return dt;                               //返回所求时间差
        }
}
```

- 第13-19行为本类的构造器，其中对物理世界以及铰链关节转动需要的一些成员变量进行了必要的初始化。
- 第28-46行为绘制一帧画面的onDrawFrame方法，其中首先获取了本帧画面与上一帧画面之间的时间差，然后将时间差累计进当前时间。接着根据上一小节介绍的两个公式计算出当前门应该转动到的角度，进而根据两帧之间门的角度差及时间差计算出角速度，然后将角速度传递给了铰链关节的旋转马达，最后绘制了两个门。
- 第50-58行为进行初始化的onSurfaceCreated方法，其中主要是创建了两个门对应的对象，并调用addHingeConstraint方法为两个门添加了铰链约束。要注意的是不动的门其质量为0，这样JBullet物理引擎就知道此刚体是静止的了。
- 第59-74行为本类中最重要的addHingeConstraint方法，所有与铰链关节有关的初始化工作都是在其中完成的。其中最重要的是第60-69行，这部分代码创建并设置了从铰链到两个物体质心的变换。
- 第76-80行为获取两帧画面间时间差的getDeltaTimeMicroseconds方法。需要特别注意的是，这里的时间指的是物理世界中的虚拟时间，不是实际机器上的时间，因此要通过JBullet中提供的Clock类的对象来获取。

> **提示**
>
> 请读者注意的是，默认情况下铰链关节的旋转轴与*Z*轴是同方向的，而本案例中希望旋转轴与*Y*轴同方向，因此需要在变换中将轴旋转90°（即BulletGlobals.SIMD_HALF_PI）。

17.8　滑动关节

上一小节介绍了铰链关节的原理与使用，本小节将介绍另一种关节——滑动关节的原理与使用。现实世界中的很多机械结构都可以使用滑动关节来模拟，如龙门吊车、叉车等。

17.8.1　滑动关节的基本知识

介绍具体的案例之前，首先应该了解一下滑动关节的基本知识。滑动关节是一种仅有一个平移和旋转自由度的关节，连接到滑动关节上的物体只能在该自由度上进行滑动或者转动，其基本原理如图17-38所示。

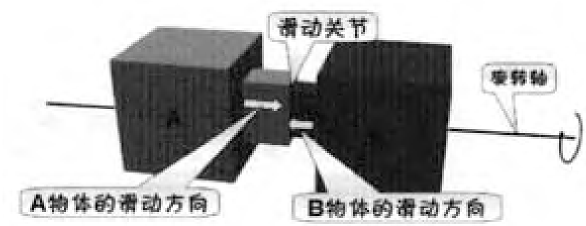


图17_38　滑动关节原理图

从图17-38中可以看出，连接到滑动关节上的物体仅能在关节确定的自由度上（也就是轴上）滑动或转动，如现实生活中的螺丝或螺母就是如此。实现滑动关节需要用到的约束类为SliderConstraint，其构造器及常用方法如表17-16所列。

表17-16　SliderConstraint类的构造器和常用方法

构造器或方法签名	含　义	类型
public SliderConstraint(RigidBody rbA, RigidBody rbB, Transform frameInA, Transform frameInB , boolean useLinearReferenceFrameA)	创建滑动关节对应的约束对象。参数rbA代表第一个要添加约束的刚体，参数rbB代表第二个要添加约束的刚体，参数frameInA代表从约束位置到A刚体质心位置的变换，参数frameInB代表从约束位置到B一刚体质心位置的变换。参数useLinearReferenceFrameA表示两个刚体与两个约束之间的对应关系,若其为true,则rbA与framelnA对应,rbB与framelnB对应，否则rbA与framelnB对应，rbB与framelnA对应	构造器
public void	设置滑动关节滑动距离的上限，	方法

setUpperLinLimit (float upperLimit)	参数upperLimit为滑动距离的上限值	
public void setLowerLinLimit (float lowerLimit)	设置滑动关节滑动距离的下限，参数lowerLimit为滑动距离的下限值	方法
public float getUpperLinLimit()	获取滑动距离的上限值	方法
public float getLowerLinLimit()	获取滑动距离的下限值	方法
public void setLowerAngLimit (float lowerLimit)	设置滑动关节转动角度的下限，参数lowerLimit为转动角度的下限值	方法
public void setUpperAngLimit (float upperLimit)	设置滑动关节转动角度的上限，参数upperLimit为转动角度的上限值	方法
public float getLowerAngLimit()	获取转动角度的下限值	方法
public float getUpperAngLimit()	获取转动角度的上限值	方法
public float setDampingDirLin (float dampingDirLin)	设置关节的滑动阻尼系数	方法
public float setDampingDirAng (float dampingDirAng)	设置关节的转动阻尼系数	方法
public float getDampingDirLin()	获取关节的滑动阻尼系数	方法
public float getDampingDirAng()	获取关节的转动阻尼系数	方法
public void setPoweredLinMotor (boolean onOff)	设置是否启动滑动运动对应的马达,参数onOff为true时启动,否则不启动	方法
public void	设置滑动运动驱动马达的最大力	方法

setMaxLinMotorForce (float maxLinMotorForce)	量值,参数maxLinMotorForce为要设置的最大力量值	
public void setTargetLinMotorVelocity (float targetLinMotorVelocity)	设置滑动运动驱动马达的速度,参数targetLinMotorVelocity为要设置的速度值	方法
public void setPoweredAngMotor (boolean onOff)	设置是否启动转动运动对应的马达,参数onOff为true时启动,否则不启动	方法
public void setMaxAngMotorForce (float maxAngMotorForce)	设置转动运动驱动马达的最大力量值,参数maxAngMotorForce为要设置的最大力量值	方法
public void setTargetAngMotorVelocity (float targetAngMotorVelocity)	设置转动运动驱动马达的速度,参数targetAngMotorVelocity为要设置的速度值	方法

17.8.2　案例效果图

介绍案例的具体开发之前,应该首先了解一下本节案例的运行效果,具体情况如图17-39至图 17-42这4幅图所示。

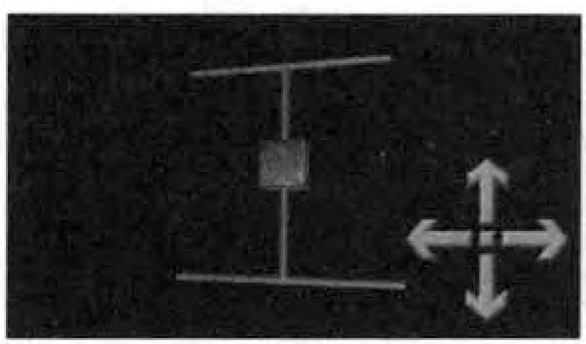


图17-39　案例初始状态

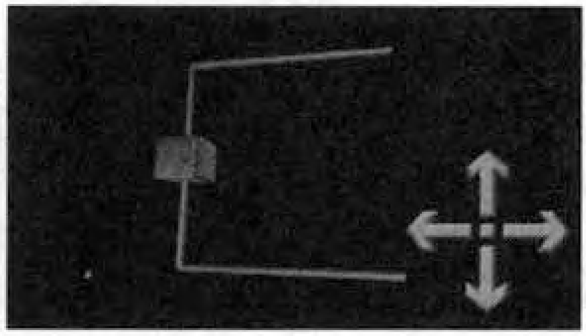


图17-40　向左滑动后的效果图

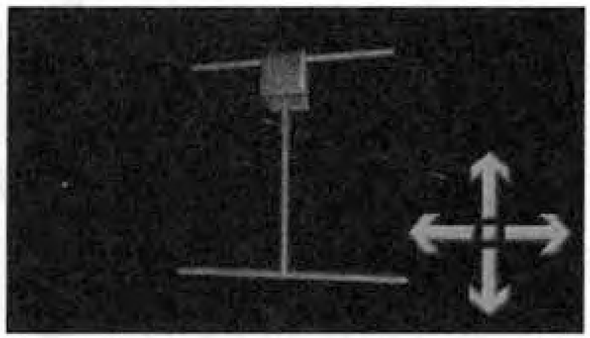


图17-41　向上滑动后的效果图

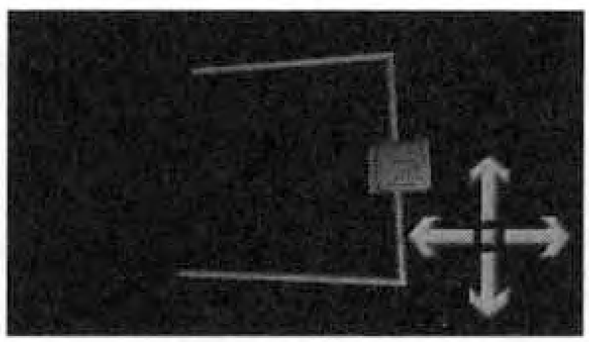


图17-42　向右滑动后的效果图

提示

案例运行时用手指触摸屏幕右下侧上（下）方向的箭头，箱子会向上（下）运动；用手指触摸屏幕右下侧左（右）方向的箭头，中间的竖杆会左（右）运动。

从上述4幅图中可以分析出，箱子所在的竖杆与一个左右方向的滑动关节相连，箱子与一个上下方向的滑动关节相连，形成了一个类似于龙门吊的组合机械结构。

17.8.3 3D界面渲染类——MySurfaceView

通过前面的几个小节的介绍，读者应该对滑动关节及本节案例的功能有了一定的了解，下面将进一步介绍本节案例的开发。由于本节案例中很多类的代码与前面很多案例中的非常相似，因此这里重点介绍使用滑动关节的有代表性的部分，其内容如下。

（1）首先给出的是MySurfaceView类中创建相关刚体的initRigidBody方法，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_6/src/com.bn.Sample17_6目录下的MySurface View.java。

```
public void initRigidBody(){                          //创建刚体的方法
  cubeBody = new AddRigidBody(boxShape,1f,dynamicsWorld, 0.0f, 0.8f,
    new Vector3f(0,Ceiling_Height-5,0),true);   //创建立方体刚体(对应于场景中的箱子)
  stickFBSliderBody = new AddRigidBody(stickFBSliderShape, 1f, dynamicsWorld, 0.0f, 0.8f,
    new Vector3f(0,Ceiling_Height-5,0),true);   //创建圆柱刚体(对应于场景中的竖杆)
  stickLRFSliderBody = new AddRigidBody(stickLRSliderShape, 0, dynamicsWorld, 0.0f, 0.8f,
    new Vector3f(0,Ceiling_Height-5,-Stick_Length),true);//创建圆柱刚体(对应于上侧的横杆)
  stickLRNSliderBody = new AddRigidBody(stickLRSliderShape, 0, dynamicsWorld, 0.0f,
0.8f,
    new Vector3f(0,Ceiling_Height-5,Stick_Length),true); //创建圆柱刚体(对应于下侧的横杆)
}
```

- 上述方法的一个重要任务是创建了场景中各个物体对应的刚体对象。
- 另一方面需要注意的是要将各个刚体的初始位置设置到位，即将3根圆柱刚体摆放成“工”字形（见图17-39），将箱子摆放到竖杆的中心。

提示

若在为刚体添加约束之前没有将刚体摆放到位，如两个刚体摆放的位置相距很远。那么在添加约束之后，两个刚体会因约束的作

用而相向运动，这会在程序初始运行时体现出来，而且有可能导致程序最终的运行效果不正常，所以提前将各刚体摆放到位是十分必要的。

（2）接着给出的是初始化物理世界，并给场景中的相关刚体添加滑动关节约束的initWorld方法，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_6/src/com.bn.Sample17_6目录下的MySurfaceView.java。

```
public void initWorld(){                                    //初始化物理世界的方法
    ……//此除省略了部分与前面几个案例相似的代码，需要的读者请参考光盘
    initRigidBody();                                        //创建刚体
    //下面的代码添加箱子与场景中竖杆之间的滑动关节约束
    Vector3f originA = new Vector3f(0, 0, 0);               //从约束到竖杆质心的平移变换信息
    Vector3f originB = new Vector3f(0, 0, 0);               //从约束到箱子质心的平移变换信息
    addSliderConstraint(0,stickFBSliderBody.body,           //调用方法添加滑动关节约束
      cubeBody.body,BulletGlobals.SIMD_PI/2,originA,originB,true);//BulletGlobals.
      SIMD_PI/2
    //下面的代码添加竖杆与上侧横杆之间的滑动关节约束
    originA = new Vector3f(0,0,0);                      //从约束到上侧横杆质心的平移变换信息
        originB = new Vector3f(0, 0, -Stick_Length);    //从约束到竖杆质心的平移变换信息
        addSliderConstraint(1,stickLRFSliderBody.body,  //调用方法添加滑动关节约束
          stickFBSliderBody.body,0,originA,originB,true);
        //下面的代码添加竖杆与下侧横杆之间的滑动关节约束
        originA = new Vector3f(0,0,0);                  //从约束到下侧横杆质心的平移变换信息
        originB = new Vector3f(0, 0, Stick_Length); //从约束到竖杆质心的平移变换信息
        addSliderConstraint(2,stickLRNSliderBody.body,  //调用方法添加滑动关节约束
          stickFBSliderBody.body,0,originA,originB,true);
}
```

说明

上述方法中主要是在相关刚体间分别添加了3个滑动关节约束，分别为：箱子与场景中竖杆之间的滑动关节约束、竖杆与上侧横杆之间的滑动关节约束以及竖杆与下侧横杆之间的滑动关节约束。

（3）下面给出的是具体实现给场景中的相关刚体添加滑动关节约束的addSliderConstraint方法，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_6/src/com.bn.Sample17_6目录下的MySurfaceView.java。

```
public void addSliderConstraint(int index,RigidBody ra,RigidBody
    rb,float angle,Vector3f originA,Vector3f originB,boolean force){
    Transform localA = new Transform();     //创建变换对象A(表示从变换到刚体 ra 质心的变换)
    Transform localB = new Transform();     //创建变换对象B(表示从变换到刚体 rb 质心的变换)
    localA.setIdentity();                   //初始化变换对象A
    localB.setIdentity();                   //初始化变换对象B
    MatrixUtil.setEulerZYX(localA.basis, 0,angle, 0 );  //设置变换对象A的旋转部分
    localA.origin.set(originA);                          //设置变换对象A的平移部分
    MatrixUtil.setEulerZYX(localB.basis, 0, angle, 0);  //设置变换对象B的旋转部分
    localB.origin.set(originB);                          //设置变换对象B的平移部分
        if(index==0){//若索引号为0，则添加箱子与场景中竖杆之间的滑动关节约束

          sliderFB = new SliderConstraint(ra, rb, localA, localB, force);
                                                        //创建滑动关节约束对象
          sliderFB.setLowerLinLimit(-Stick_Length);     //滑动的下限
          sliderFB.setUpperLinLimit(Stick_Length);      //滑动的上限
          sliderFB.setLowerAngLimit(0);                 //转动的下限
          sliderFB.setUpperAngLimit(0);                 //转动的上限
          sliderFB.setDampingDirLin(0.05f);             //设置滑动阻尼系数
          dynamicsWorld.addConstraint(sliderFB,true);   //将约束添加进物理世界
          sliders[index]=sliderFB;                      //将约束放进约束数组
        }
        if(index==1){//若索引号为1，则添加竖杆与上侧横杆之间的滑动关节约束
          sliderLRF=new SliderConstraint(ra, rb, localA, localB, force);
                                                        //创建滑动关节约束对象
          sliderLRF.setLowerLinLimit(-Stick_Length);    //滑动的下限
          sliderLRF.setUpperLinLimit(Stick_Length);     //滑动的上限
          sliderLRF.setLowerAngLimit(0);                //转动的下限
          sliderLRF.setUpperAngLimit(0);                //转动的上限
          sliderLRF.setDampingDirLin(0.5f);             //设置滑动阻尼系数
            dynamicsWorld.addConstraint(sliderLRF,true); //将约束添加进物理世界
            sliders[index]=sliderLRF;                   //将约束放进约束数组
        }
        if(index==2){//若索引号为2，则添加竖杆与下侧横杆之间的滑动关节约束
          sliderLRN=new SliderConstraint(ra, rb, localA, localB, force);
                                                        //创建滑动关节约束对象
          sliderLRN.setLowerLinLimit(-Stick_Length);    //滑动的下限
          sliderLRN.setUpperLinLimit(Stick_Length);     //滑动的上限
          sliderLRN.setLowerAngLimit(0);                //转动的下限
          sliderLRN.setUpperAngLimit(0);                //转动的上限
          sliderLRN.setDampingDirLin(0.5f);             //设置滑动阻尼系数
          dynamicsWorld.addConstraint(sliderLRN,true);  //将约束添加进物理世界
          sliders[index]=sliderLRN;                     //将约束放进约束数组
        }
    }
```

说明

上述addSliderConstraint方法中根据索引编号的不同分别创建了箱子与场景中竖杆之间的滑动关节约束、竖杆与上侧横杆之间的滑动关节约束以及竖杆与下侧横杆之间的滑动关节约束。

（4）接着给出的是驱动箱子与竖杆之间关节滑动的slideFB方法，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_6/src/com.bn.Sample17_6目录下的MySurfaceView.java。

```
public void slideFB(float mulFactor){
    sliding=true;                                    //设置滑动标志位为 true
    sliderFB.getRigidBodyB().activate();             //激活箱子刚体
    currIndex=0;                                     //设置当前动作索引号为 0
    sliderFB.setPoweredLinMotor(true);               //启动关节对应的马达
    sliderFB.setMaxLinMotorForce(1.0f);              //设置马达的滑动驱动力
    sliderFB.setTargetLinMotorVelocity(5.0f*mulFactor); //设置马达的滑动驱动速度
}
```

说明

上述方法非常简单，仅仅是设置并启动了驱动箱子与竖杆之间关节滑动的马达。

（5）下面给出的是驱动竖杆与上下两侧横杆之间关节滑动的slideLR方法，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_6/src/com.bn.Sample17_6目录下的MySurfaceView.java。

```
public void slideLR(float mulFactor){
    sliding=true;                                  //设置滑动标志位为 true
    sliderLRF.getRigidBodyB().activate();          //激活竖杆刚体
    currIndex=1;                                   //设置当前动作索引号为 0
    sliderLRF.setPoweredLinMotor(true);            //启动竖杆与上侧横杆间关节对应的马达
    sliderLRF.setMaxLinMotorForce(5.0f);           //设置马达的滑动驱动力
    sliderLRF.setTargetLinMotorVelocity(5.0f*mulFactor);     //设置马达的滑动驱动速度
    sliderLRN.setPoweredLinMotor(true);            //启动竖杆与上侧横杆间关节对应的马达
    sliderLRN.setMaxLinMotorForce(5.0f);           //设置马达的滑动驱动力
        sliderLRN.setTargetLinMotorVelocity(5.0f*mulFactor);//设置马达的滑动驱动速度
    }
```

说明

上述方法与前面介绍的slideFB方法类似，仅仅是设置并启动了驱动竖杆与上下两侧横杆之间关节滑动的马达。

（6）最后给出的是停止所有关节滑动的stopSlide方法，其代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_6/src/com.bn.Sample17_6目录下的MySurfaceView.java。

```
public void stopSlide(){
    sliding=false;                                           //设置滑动标志位为 false
    sliders[currIndex].setPoweredLinMotor(false);            //关闭当前动作编号对应的马达
    sliders[currIndex].setMaxLinMotorForce(0.0f);            //设置马达的滑动驱动力为 0
    sliders[currIndex].setTargetLinMotorVelocity(0.0f);      //设置马达的滑动驱动速度为 0
    if(currIndex==1){                                        //若动作编号为 1
      sliders[currIndex+1].setPoweredLinMotor(false);        ////关闭下一动作编号对应的马达
      sliders[currIndex+1].setMaxLinMotorForce(0.0f);        //设置马达的滑动驱动力为 0
      sliders[currIndex+1].setTargetLinMotorVelocity(0.0f);     //设置马达的滑动驱动速度为 0
        }
    }
```

说明

上述方法非常简单，主要功能为根据当前的动作编号关闭相应关节的驱动马达。需要特别注意的是，若动作编号为1，也就是对应的动作是左右滑动时，需要关闭上下两侧的马达。

17.9　六自由度关节

前面两节分别介绍了铰链关节与滑动关节，这两种关节各自可以用来模拟不同的机械结构。本节将介绍JBullet中能力最强的一种关节——六自由度关节。

17.9.1　六自由度关节的基本知识

顾名思义，六自由度关节有6个不同的自由度，包括3个平移（滑动）自由度以及3个旋转自由度。通过其可以模拟现实世界中很多动物的骨骼关节以及机械结构，例如：肘关节、颈关节、机械手臂等。由于此种关节具有3D世界中全部的6个自由度，因此具有很大的灵活性。

通过锁死或限制此种关节的某个或某几个自由度，可以轻松地模仿其他类型的关节。例如：仅允许一个旋转自由度，锁死其他5个自由度，就可以模仿前面17.7节介绍过的铰链关节。

六自由度关节的基本实现思路为：给刚体指定一个连接点，刚体本身可以绕3个轴旋转，同时可以在3个轴的方向上进行平移，但旋转与平移都受到连接点的限制。实现六自由度关节需要用到的约束为Generic6DofConstraint，其构造器及常用方法如表17-17所列。

表17-17　Generic6DofConstraint类的构造器和常用方法

构造器或方法签名	含　义	类型
Generic6DofConstraint (RigidBody rbA, RigidBody rbB, Transform frameInA, Transform frameInB, boolean useLinearReferenceFrameA)	创建一个连接两个刚体的六自由度关节。参数rbA代表第一个要添加约束的刚体，参数rbB代表第二个要添加约束的刚体，参数frameInA代表从约束位置到A刚体质心位置的变换，参数frameInB代表从约束位置到B刚体质心位置的变换。参数useLinearReferenceFrameA表示两个刚体与两个约束之间的对应	构造器

	关系，若其为true，则rbA与frameInA对应，rbB与frameInB对应，否则rbA与frameInB对应，rbB与frameInA对应	
public void setLinearLowerLimit (Vector3f linearLower)	设置关节3个自由度（3个轴）滑动距离的下限，参数linearLower为滑动距离下限值的3个分量	方法
public void setLinearUpperLimit (Vector3f linearUpper)	设置关节3个自由度（3个轴）滑动距离的上限，参数linearUpper为滑动距离上限值的3个分量	方法
public void setAngularLowerLimit (Vector3f angularLower)	设置关节3个自由度（3个轴）转动角度的下限，参数angularLower为转动角度下限值的3个分量	方法
public void setAngularUpperLimit (Vector3f angularUpper)	设置关节3个自由度（3个轴）转动角度的上限，参数angularUpper为转动角度上限值的3个分量	方法
public float getAngle(int axis_index)	获取指定轴的旋转角度（基于欧拉角的计算方法）	方法
public Transform getCalculated- TransformA (Transform out)	获取关节连接的第一个刚体在世界坐标系中的变换	方法
public Transform getCalculated- TransformB(Transform out)	获取关节连接的第二个刚体在世界坐标系中的变换	方法
public void setLimit(int axis,float lo,float hi)	设置指定编号自由度的下限与上限，前3个为滑动自由度，后3个为转动自由度。参数lo为下限值，参数hi为上限值。此方法组合使用可以代替前面的setLinearLowerLimit、setLinearUpperLimit、setAngularLowerLimit与setAngularUpperLimit等4个方法	方法

17.9.2 案例的运行效果

介绍案例的具体开发之前，应该首先了解一下本节案例的运行效果，具体情况如图17-43至图17-46这4幅图所示。

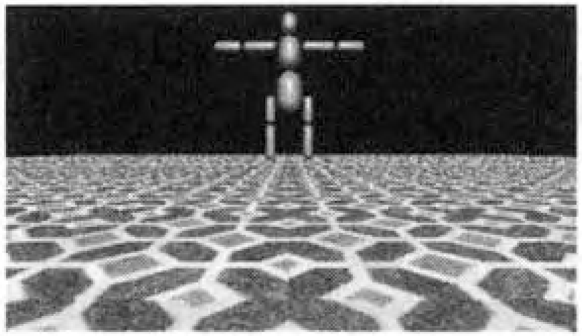


图17-43 人偶初始状态

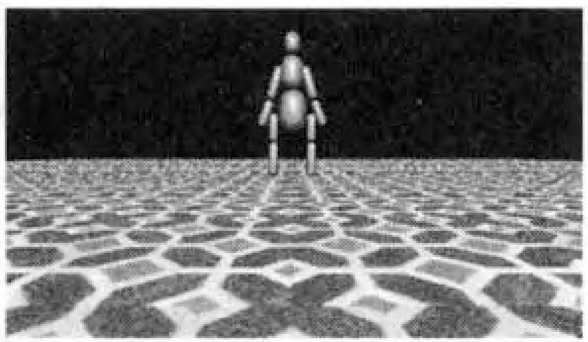


图17-44 人偶下落着地

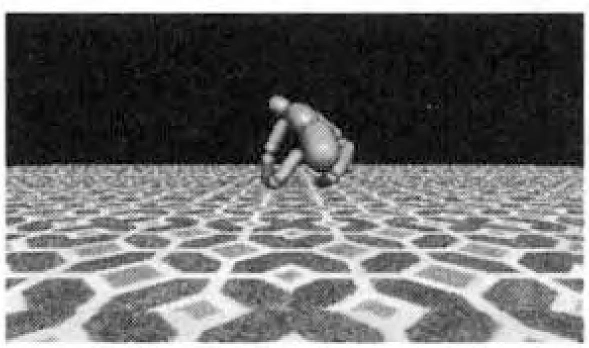


图17-45 人偶身子逐渐倒下

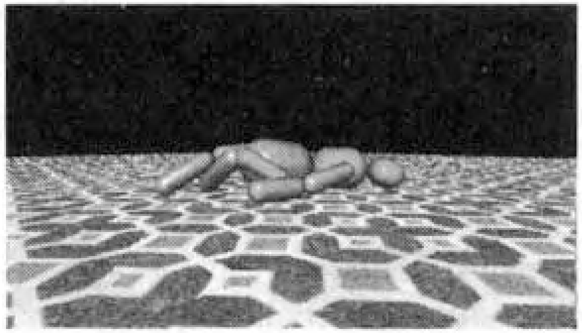


图17-46　人偶完全倒下

说明

从图17-43至图17-46这4幅图中可以看出，本节案例的场景中有一个由11个不尽相同的胶囊形状搭建而成的人偶从空中下落，搭建人偶的胶囊之间都由六自由度关节相连。

下面简单说明一下本节案例运行时的操作方法以及人偶在不同状态下的颜色变化情况，具体内容如下所列。

- 用手指触摸屏幕左上角（右上角），摄像机绕场景顺时针（逆时针）转动。
- 用手指触摸屏幕左下角（右下角），摄像机前进（后退）。
- 当手指触摸到人偶的任何一个组成部分时，此组成部分变为绿色，并随手指的滑动而移动。通过此操作可以将人偶按照希望的方向拖动。
- 当人偶完全静止时，其将整体变为蓝绿色。

提示

由于本书插图采用灰度印刷，因此人偶颜色的变化可能很难从插图中看出，这时请读者自行采用真机运行本案例。另外，由于本案例物理模拟需要的计算量较大，在一些低端的机型上运行可能速度不是很快。但一般的机型上还是没有问题的，笔者在摩托罗拉的里程碑2、ME860、ME865以及华为的U8800+等机型上运行都还比较流畅。

17.9.3　人偶类——Doll

本案例中的很多代码与前面不少案例中的非常类似，这里不再赘述。本小节将介绍本案例中最重要的人偶类——Doll，其主要负责通过六自由度关节搭建人偶。从上一小节的运行效果图中可以看出，人偶身体的各部分均由胶囊形状构成，包括头、大臂、小臂、脊椎等11部分。

由于各相关部分之间均采用六自由度关节相连，故创建连接各部分关节的代码基本相同。因此这里仅详细介绍与头部以及连接到头部的脊椎相关的代码，具体内容如下。

代码位置：见随书光盘中源代码/第17章/Sample17_7/src/com.bn.Sample17_7目录下的Doll.java。

```
package com.bn.Sample17_7;                                    //声明包
……//此处省略了部分引入相关类的代码，需要的读者请参考随书光盘
public class Doll {
```

```
MySurfaceView mv;                                          //场景渲染类对象的引用
DynamicsWorld dynamicsWorld;                               //物理世界引用
//碰撞形状数组，组成人偶身体各部分的胶囊形状均存储于此数组中
CollisionShape[] bodyShapes= new CollisionShape[BodyPartIndex.BODYPART_COUNT.
ordinal()];
//刚体数组，组成人偶身体各部分的刚体均存储于此数组中，每个刚体对应一个胶囊形状
RigidBody[] rigidBodies = new RigidBody[BodyPartIndex.BODYPART_COUNT.ordinal()];
LoadedObjectVertexNormal[] bodyForDraws;                   //用于渲染身体各部分的加载物体
float mass=1;                                              //刚体的质量
float bodyCenterH=7;                                       //整个人偶中心的高度
float headR=0.5f;                                          //头部胶囊两端半球的半径
float headH=1.25f-2*headR;                                 //头部胶囊中间圆柱的长度
float headTH=1.25f;                                        //头部胶囊总长度
float spineR=0.7f;                                         //脊椎胶囊两端半球的半径
float spineH=2.0f-2*spineR;                                //脊椎胶囊中间圆柱的长度
float spineTH=2.0f;                                        //脊椎胶囊总长度
……//此处省略了其他胶囊碰撞形状的属性值，需要的读者可以参考光盘中的源代码
public Doll(MySurfaceView mv,DynamicsWorld dynamicsWorld,
  LoadedObjectVertexNormal[] bodyForDraws){                //人偶类构造器
  this.mv=mv;                                              //保存场景渲染类对象的引用
  this.dynamicsWorld=dynamicsWorld;                        //保存物理世界的引用
  this.bodyForDraws=bodyForDraws;                          //保存用于渲染的物体列表引用
  initShapes();                                            //初始化碰撞形状
  initRigidBodys();                                        //初始化刚体
}
public void initShapes(){                                  //初始化碰撞形状的方法
  //创建头部对应的胶囊碰撞形状
 bodyShapes[BodyPartIndex.BODYPART_HEAD.ordinal()] = new CapsuleShape(headR,headH);
  //创建脊椎对应的胶囊碰撞形状
 bodyShapes[BodyPartIndex.BODYPART_SPINE.ordinal()] = new CapsuleShape(spineR,spineH);
  ……//此处省略了创建构成身体其他部分胶囊碰撞形状的代码，读者可以查阅随书光盘中的源代码
}
public void initRigidBodys(){                              //初始化刚体的方法
  Transform tempTrans = new Transform();                   //创建刚体的初始变换对象
  tempTrans.setIdentity();                                 //初始化变换对象
  tempTrans.origin.set(0,bodyCenterH+spineTH+headTH/2,0); //设置头部沿 Y 轴进行平移
  rigidBodies[BodyPartIndex.BODYPART_HEAD.ordinal()]= //创建头部刚体
  addRigidBody(mass,tempTrans,bodyShapes[BodyPartIndex.BODYPART_HEAD.ordinal()]);
  tempTrans.setIdentity();                                 //初始化变换对象
 tempTrans.origin.set(new Vector3f(0,bodyCenterH+spineTH/2,0));//设置脊椎沿 Y 轴进行平移
  rigidBodies[BodyPartIndex.BODYPART_SPINE.ordinal()] =   //创建脊椎刚体
 addRigidBody(mass, tempTrans, bodyShapes[BodyPartIndex.BODYPART_SPINE.ordinal()]);
  ……//此处省略了创建身体其他部分刚体的代码，读者可以查阅随书光盘中的源代码
  Generic6DofConstraint joint6DOF;                //声明六自由度关节引用
  Transform localA = new Transform();             //创建变换对象 A(表示从变换到刚体 A 质心的变换)
  Transform localB = new Transform();             //创建变换对象 B(表示从变换到刚体 B 质心的变换)
  localA.setIdentity();                           //初始化变换对象 A
  localB.setIdentity();                           //初始化变换对象 B
```

```
51      localA.origin.set(0f, -headTH/2, 0f);  ////设置变换对象A的平移部分
52      localB.origin.set(0,spineTH/2,0);      ////设置变换对象B的平移部分
53      //创建头部和脊椎之间的六自由度约束
54      joint6DOF=new Generic6DofConstraint(rigidBodies[BodyPartIndex.BODYPART_HEAD.
        ordinal()],
55              rigidBodies[BodyPartIndex.BODYPART_SPINE.ordinal()],localA,localB,true);
56      Vector3f limitTrans = new Vector3f();  //用于存储六自由度关节旋转角度上/下限的向量
57      limitTrans.set(-BulletGlobals.SIMD_PI * 0.1f, -BulletGlobals.FLT_EPSILON,
58                     -BulletGlobals.SIMD_PI * 0.1f);    //设置旋转角度下限的3个分量
59      joint6DOF.setAngularLowerLimit(limitTrans);        //设置关节转动角度的下限
60      limitTrans.set(BulletGlobals.SIMD_PI * 0.2f, BulletGlobals.FLT_EPSILON,
61                     BulletGlobals.SIMD_PI * 0.2f);     //设置旋转角度上限的3个分量
62      joint6DOF.setAngularUpperLimit(limitTrans);        //设置关节转动角度的上限
63      dynamicsWorld.addConstraint(joint6DOF, true);      //将约束添加进物理世界
64        ……//此处省略了创建及初始化他身体部分(刚体)之间六自由度关节的代码
65    }
66    private RigidBody addRigidBody(float mass,Transform startTransform,CollisionShape
      shape)
67    {/*代码省略*/}  //此方法代码与本章前面案例中的基本相同,需要的读者请参考随书光盘
68    public void drawSelf(int checkedIndex){       //绘制人偶的方法
69      MatrixState.pushMatrix();                   //保护现场
70      for(int i=0;i<bodyForDraws.length;i++){     //遍历用于渲染身体各部分的加载物体数组
71        LoadedObjectVertexNormal lovo = bodyForDraws[i];
72        if(i==checkedIndex){                      //如果该物体(身体部分)被拾取
73          lovo.changeColor(true);                 //设置改变颜色的标志位为true
74          lovo.drawSelf(rigidBodies[i]);          //绘制该物体(身体部分)
75        }else{          //若该物体(身体部分)没有被拾取
76          lovo.changeColor(false);                //设置改变颜色的标志位为false
77          lovo.drawSelf(rigidBodies[i]);          //绘制该物体(身体部分)
78        }
79      }
80      MatrixState.popMatrix();                    //恢复现场
81    }
82 }
```

• 第28-34行为初始化胶囊碰撞形状的initShapes方法，其中创建了对应于身体各部分的11个胶囊碰撞形状对象。

• 第35-65行为基于initShapes方法创建的11个胶囊碰撞形状对象创建对应的刚体，并在相关刚体之间添加六自由度关节约束的initRigidBodys方法。这里仅给出了创建头部刚体、脊椎刚体以及在这两个刚体之间添加六自由度关节约束的代码，创建身体其他部分对应的刚体以及添加六自由度关节约束的代码与之非常类似，需要的读者请参考随书光盘中的源代码。

• 第68-81行为绘制人偶身体各部分的drawSelf方法，其实现非常简单。基本思路为：遍历用于渲染身体各部分的加载物体数组，根据各部分对应刚体的姿态情况进行绘制。

17.9.4 拾取时采用的点对点关节

从前面17.9.2小节介绍案例操作方法的内容中已经知道，本节案例不但具有物理模拟功能，还 可以通过手指触摸屏幕选中3D虚拟世界中的物体并进行拖动。通过手指触摸屏幕选中3D虚拟世界中的物体也称之为拾取，关于拾取技术的具体实现将在第19章详细介绍。

由于实现拖动的功能时用到了点对点关节，因此这里首先对其进行简要介绍，然后再介绍通过拾取实现选中、拖动的思路以及相关代码。

1．点对点关节简介

点对点关节是仅有3个旋转自由度，没有平移自由度的关节。连接到点对点关节上的物体只能任意旋转，而不能相对于关节平移，其基本原理如图17-47所示。

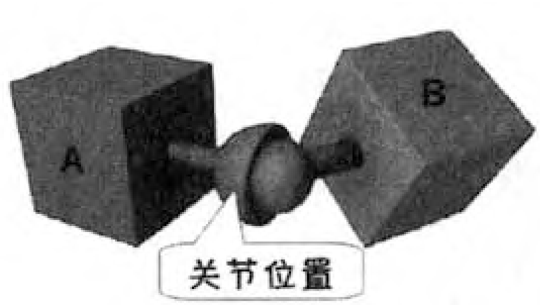


图17-47　点对点关节原理

实现点对点关节需要用到的约束为Point2PointConstraint，其构造器及常用方法如表17-18所列。

表17-18　Point2PointConstraint类的构造器和常用方法

构造器或方法签名	含　义	类型
public Point2PointConstraint (RigidBody rbA, Vector3f pivotInA)	创建添加到一个刚体上的点对点约束。参数rbA代表要添加约束的刚体，参数pivotInA为关节在此刚体坐标系中的位置	构造器
public Point2PointConstraint	创建两个刚体之间的点对点约束。参数rbA代表第一个要添加约束的刚体，	构造器

(RigidBody rbA, RigidBody rbB, Vector3f pivotInA, Vector3f pivotInB)	参数rbB代表第二个要添加约束的刚体，参数pivotInA为关节在第一个刚体坐标系中的位置，参数pivotInB为关节在第二个刚体坐标系中的位置	
public void setPivotA(Vector3 f pivotA)	设置关节在第一个刚体坐标系中的位置	方法
public void setPivotB(Vector3f pivotB)	设置关节在第二个刚体坐标系中的位置	方法
public Vector3f getPivotInA(Vector3f out)	获取关节在第一个刚体坐标系中的位置，参数out用于存储返回值	方法
public Vector3f getPivotInB(Vector3f out)	获取关节在第二个刚体坐标系中的位置，参数out用于存储返回值	方法

2．通过拾取实现选中、拖动的思路以及相关代码

首先对本案例中通过拾取技术实现拖动人偶某个组成部分的思路进行简要介绍，具体内容如下。

- 当手指在屏幕上按下时，需要首先通过拾取技术判断出有没有人偶的某个组成部分被选中，哪个部分被选中。
- 确定选中部分后，新建一个点对点关节，此关节中包含选中部分对应的刚体。
- 随着手指在屏幕上的移动更新点对点关节的位置，以实现拖动。
- 当手指抬起时，将此点对点关节从物理世界中删除。

上述思路的实现主要涉及两个类中的相关代码，具体内容如下。

（1）首先涉及的是MySurfaceView类中的onTouchEvent以及changeObj方法，具体代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_7/src/com.bn.Sample17_7目录下的MySurface

View. java。

```
public boolean onTouchEvent(MotionEvent event){ //处理触控事件的方法
    x=event.getX();y=event.getY();              //获取触控点的位置
    switch(event.getAction()){                  //根据不同的动作类型执行不同的操作
        case MotionEvent.ACTION_DOWN:           //按下的动作
          mPreviousX=event.getX();              //记录按下动作对应的位置
          mPreviousY=event.getY();
          preNanoTime=System.nanoTime();        //获取当前系统时间
          checkedIndex=-1;                      //设置选中物体索引(-1 表示没有选中物体)
          float[] AB=IntersectantUtil.calculateABPosition( //计算拾取用射线的起点与终点
                    x,                                      //触控点 X 坐标
                    y,                                      //触控点 Y 坐标
                    Sample17_7_Activity.screenWidth,        //屏幕宽度
                    Sample17_7_Activity.screenHeight,       //屏幕高度
                    left,                                   //近平面 left 值
                    top,                                    //近平面 top 值
                    near,                                   //近平面距离
                    far                                     //远平面距离
            );20
            MyVector3f start = new MyVector3f(AB[0], AB[1], AB[2]);  //拾取射线的起点A
            MyVector3f end = new MyVector3f(AB[3], AB[4], AB[5]);//拾取射线的终点 B
            MyVector3f dir = end.minus(start);     //拾取射线的长度和方向
            int tmpIndex=-1;                    //用于记录距离 A 点最近物体的索引值
            float minTime=1;                    //用于记录列表中物体与射线相交的最短时间
            for(int i=0;i<lovnList.size();i++){      //遍历列表中的物体
              AABB3 box = lovnList.get(i).getCurrBox();  //获得物体的 AABB 包围盒
              float t = box.rayIntersect(start, dir, null);   //计算相交时间
              if (t <= minTime) {                 //若时间小于原最短时间
                  minTime = t;                    //更新最短时间
                  tmpIndex = i;                   //更新最近物体索引
            }}
            checkedIndex=tmpIndex;                //记录最近(选中)物体索引
            changeObj(checkedIndex);              //改变被选中物体
          break;
          case MotionEvent.ACTION_MOVE:           //移动的动作
            //移动距离大于指定的阈值，才认为是真正的移动
            if(x-mPreviousX>=10.0f||x-mPreviousX<=-10.0f){ isMoveFlag=true;}
            if(isMoveFlag){                       //若构成移动
             if(checkedIndex!=-1){                //若被选中物体索引不为-1
               LoadedObjectVertexNormal lovo=lovnList.get(checkedIndex);//获取选中物体
               lovo.isPicked=true;                //设置此物体的选中标志位为 true
               //计算新的触控点在世界坐标系中的位置
               float[] nearXY=IntersectantUtil.calculateABPosition(
                  x,y,Sample17_7_Activity.screenWidth,Sample17_7_Activity.
                  screenHeight,
                  left,top,near,far);
               //计算上一触控点在世界坐标系中的位置
               float[] nearPreXY = IntersectantUtil.calculateABPosition(
                  mPreviousX,mPreviousY,
                  Sample17_7_Activity.screenWidth,Sample17_7_Activity.screenHeight,
                  left,top,near,far);
               //获取被选中物体所属点对点关节在刚体坐标系中的位置
```

```java
                Vector3f currPivot = lovo.p2p.getPivotInB(new Vector3f());
                Vector3f dir1 = new Vector3f(     //计算出拖动方向向量3个分量的值
                   nearXY[0]-nearPreXY[0], nearXY[1]-nearPreXY[1],nearXY[2]
                   -nearPreXY[2]);
                float vFactor = 0.5f;dir1.set(dir1.x*vFactor,dir1.y*vFactor,dir1.z
                *vFactor);
                currPivot.add(dir1);              //计算出拖动后的关节位置
                lovo.p2p.setPivotB(currPivot);    //设置拖动后的关节位置
            }}
          break;
          case MotionEvent.ACTION_UP:             //抬起的动作
            isMoveFlag=false;                     //将移动标志位设置为false
            if(checkedIndex!=-1){                 //若有物体被选中
              LoadedObjectVertexNormal lovo = lovnList.get(checkedIndex);
              lovo.removePickedConstraint();      //删除用于拖动的点对点关节
              checkedIndex=-1;                    //选中物体索引设置为-1，表示无选中物体
            }
            if(checkedIndex==-1 && currArea!=Area.NONE){   //若没有选中物体
              currArea=Area.NONE; areaTouch=false; upRadians=tempRadians;
            }
          break;
      }
      return true;
  }
  public void changeObj(int index){                 //根据索引改变选中物体
      if(index != -1){//如果有物体被选中
          LoadedObjectVertexNormal lovo = lovnList.get(index);
          lovo.body.activate();
          lovo.addPickedConstraint();
      }
      else{/*此处省略了部分与摄像机绕场景旋转以及前进后退相关的代码，与拾取无关*/}
  }
```

提示

上述代码主要是实现了前面介绍的思路，其中用到了与拾取相关的一些工具类，如IntersectantUtil。这些内容将在第19章介绍3D拾取技术时详细介绍，这里读者简单了解即可。

（2）接着涉及的是物体类LoadedObjectVertexNormal中的addPickedConstraint以及removePickedConstraint方法，具体代码如下。

代码位置：见随书光盘中源代码/第17章/Sample17_7/src/com.bn.Sample17_7目录下的LoadedObjectVertexNormal.java。

```
1 public void addPickedConstraint(){
2     //创建用于拖动的点对点关节，此关节位置在被拖动物体坐标系的原点
3     p2p = new Point2PointConstraint(body, new Vector3f(0,0,0));
4     mv.dynamicsWorld.addConstraint(p2p, true);          //将关节添加进物理世界
5     this.isPicked=true;                                  //将物体的选中标志设置为 true
6 }
7 public void removePickedConstraint(){
8     if(p2p!=null){                                      //若有用于拖动的关节
9         mv.dynamicsWorld.removeConstraint(p2p);         //将关节从物理世界中删除
10        }
11        this.isPicked=false;                            //将物体的选中标志设置为 false
12    }
```

- 第1-6行为创建用于拖动的点对点关节并将其添加进物理世界的addPickedConstraint方法。
- 第7-12行为将用于拖动的点对点关节从物理世界中删除的removePickedConstraint方法。

> **提示**
>
> 上述两个方法中均根据情况设置了物体的选中标志，此标志用于帮助确定物体绘制时采用的颜色。若物体被选中则采用绿色绘制，否则不采用绿色绘制。

17.10　本章小结

本章主要介绍了JBullet 3D物理引擎的使用，通过本章的学习，读者应该可以基于JBullet开发出非常真实的物理仿真效果。另外读者应该注意的是，物理引擎本身博大精深，本章仅仅是起到了入门的作用，有兴趣的读者应该进一步深入研究。

第18章

传感器应用的开发

Android智能手机有很多津津乐道的特性，其中最吸引人特性的之一便是传感器了。传感器是一种特殊的电子装置，依据类型的不同能够探测外界各种不同物理量的变化。Android平台上支持多种传感器，主要包括：加速度传感器、磁场传感器、光传感器、温度传感器、接近传感器以及姿态传感器等，本章将按照上述顺序对Android平台下的各种传感器一一进行介绍。

18.1　基本的开发流程

虽然Android平台下有很多不同类型的传感器，这些传感器在应用程序开发中的使用细节各有不同，但无论哪种传感器其开发的基本流程都是一致的。因此在对Android平台下各种传感器进行详细介绍之前，首先对传感器的总体开发流程进行一下简要介绍，主要步骤如下。

1．获取SensorManager对象

传感器应用开发的第一步是获取SensorManager对象，具体代码如下。

```
SensorManager mySensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
```

> **说明**
>
> 从上述代码中可以看出，获取SensorManager对象的方法很简单，只要调用API中提供的Context对象（Activity对象、Service对象都是）下的getSystemService方法即可。调用时传入的参数SENSOR_SERVICE是Context类下的常量。

2．获取Sensor对象

获取了SensorManager对象后就可以通过其getDefaultSensor方法获取某种具体类型的传感器对象了。调用getDefaultSensor方法时需要传入一个常量描述需要获取的传感器类型，常用的几个常量如下所列。

- Sensor.TYPE_ACCELEROMETER（加速度传感器）。
- Sensor.TYPE_MAGNETIC_FIELD（磁场传感器）。
- Sensor.TYPE_LIGHT（光传感器）。
- Sensor.TYPE_TEMPERATURE（温度传感器）。
- Sensor.TYPE_PROXIMITY（接近传感器）。
- Sensor.TYPE_ORIENTATION（姿态传感器）。

获取传感器对象，并获取传感器各方面描述性信息的基本代码如下。

```
//获取指定类型的传感器对象(此段示例代码中获取的是加速度传感器)
Sensor myAccelerometer=mySensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
StringBuffer strb=new StringBuffer();            //创建 StringBuffer 对象
strb.append("\n 名称: ");
strb.append(myAccelerometer.getName());          //获取传感器名称
strb.append("\n 耗电量(mA): ");
strb.append(myAccelerometer.getPower());         //获取此传感器的耗电量，以毫安(mA)为单位
strb.append("\n 类型编号 : ");
strb.append(myAccelerometer.getType());          //获取传感器类型编号
    strb.append("\n 制造商: ");
    strb.append(myAccelerometer.getVendor());    //获取传感器的制造商
    strb.append("\n 版本: ");
    strb.append(myAccelerometer.getVersion());   //获取传感器版本
    strb.append("\n 最大测量范围: ");
    strb.append(myAccelerometer.getMaximumRange());//获取传感器的最大测量范围(量程)
```

3．实现SensorEventListener接口

成功地获取了传感器对象后，下面要做的工作是向传感器注册监听器。注册了监听器后，当对应传感器测量的物理量发生变化时，系统会回调监听器中的特定方法。因此在注册监听器之前首先要开发出实现了SensorEventListener接口的监听器，主要是实现SensorEventListener接口中的两个方法。

（1）onAccuracyChanged方法。

此方法的签名为“public void onAccuracyChanged（Sensor sensor, int accuracy）”，其在传感器的精度发生变化时被回调，第二个参数为当前的精度。从参数类型中可以看出，精度的类型是整数。实际开发中精度有4种可能的取值，都是SensorManager类下属的常量，按照精度由高到低的顺序如下所列。

- SENSOR_STATUS_ACCURACY_HIGH。
- SENSOR_STATUS_ACCURACY_MEDIUM。
- SENSOR_STATUS_ACCURACY_LOW。
- SENSOR_STATUS_UNRELIABLE。

（2）onSensorChanged方法。

此方法的签名为“public void onSensorChanged（SensorEvent event）”，其在传感器所测量的物理

量发生变化时被回调，传入的参数为传感器事件对象的引用。通过此引用可以获取当前传感器测量值的数组values。

根据传感器类型的不同，values数组的长度也不尽相同，例如，加速度传感器对应的values数组长度为3，其中的3个元素分别代表当

前加速度在*X* 轴、*Y* 轴、*Z* 轴上的分量，如下面的代码所示。

```
//开发实现了 SensorEventListener 接口的传感器监听器
private SensorEventListener mySensorListener = new SensorEventListener(){
```

```
public void onAccuracyChanged(Sensor sensor, int accuracy){/*精度变化时的处理代码，省略*/}
public void onSensorChanged(SensorEvent event){          //重写 onSensorChanged 方法
    float []values=event.values;                         //获取 3 个轴方向上的加速度值
    tvX.setText("x 轴方向上的加速度为："+values[0]);      //显示 x 轴方向上的加速度值
    tvY.setText("y 轴方向上的加速度为："+values[1]);      //显示 y 轴方向上的加速度值
    tvZ.setText("z 轴方向上的加速度为："+values[2]);      //显示 z 轴方向上的加速度值
}};
```

4．注册与注销监听器

完成了监听器的开发后就可以注册监听器了，这项工作一般在Activity中的onResume方法中进行，基本代码如下。

```
protected void onResume(){                               //重写 onResume 方法
    super.onResume();
    mySensorManager.registerListener(                    //注册监听器
            mySensorListener,                            //监听器引用
            myAccelerometer,                             //被监听的传感器引用
            SensorManager.SENSOR_DELAY_NORMAL            //传感器采样的频率
);}
```

> **提示**
>
> 传感器采样的频率都是用SensorManager类下属的常量来表示的，一共有4种选择：SENSOR_DELAY_FASTEST（最快频率）、SENSOR_DELAY_UI（适合普通用户界面的频率）、SENSOR_DELAY_GAME（适合游戏的频率）、SENSOR_DELAY_NORMAL（默认频率，适合屏幕横竖状态的自动切换）。

由于传感器是比较耗电的设备之一，因此当不使用时应该及时注销监听器，减少耗电量。注销监听器的工作一般在Activity的onPause方法中进行，基本代码如下。

```
protected void onPause(){                                          //重写onPause方法
    super.onPause();
    mySensorManager.unregisterListener(mySensorListener);//注销监听器
}
```

提示

本节给出的示例代码均来自加速度传感器案例Sample18_1，因此当下一小节介绍加速度传感器的开发时就不再给出重复的代码了。另外，本章给出的传感器应用程序案例需要真机上有对应的硬件才能正常运行，有部分机型没有某些传感器，案例程序可能不能正常运行。

18.2　加速度传感器

本节主要介绍加速度传感器的使用，首先给出加速度传感器的一些基本知识，然后给出一个简单的案例Sample18_1。

18.2.1　加速度传感器简介

加速度传感器用于感知手机的加速度，其返回值是一个长度为3的数组（values），数组中的3个元素（values[0]～values[2]）分别代表手机的加速度在*X* 轴、*Y* 轴、*Z* 轴上的分量，单位均为m/s^2（米/平方秒），具体情况如下。

- values[0]：*x* 轴方向的加速度减去重力加速度在*x* 轴上的分量。
- values[1]：*y* 轴方向的加速度减去重力加速度在*y* 轴上的分量。
- values[2]：*z* 轴方向的加速度减去重力加速度在*z* 轴上的分量。

上面的介绍中用到了基于手机的空间坐标系，这里简单介绍一下，基本情况如图18-1所示。



图18-1　手机空间坐标系

说明

从图18-1中可以看出，整个空间坐标系的原点位于手机屏幕的左下角。*X* 轴平行于屏幕的短边，从左至右；*Y* 轴平行于屏幕的长边，从下至上；*Z* 轴垂直于屏幕与*X* 轴与*Y* 轴正交。另外需要注意的是坐标轴是绑定在手机上的，也就是说坐标轴与手机之间的关系不随着手机姿态的变化而变化。

下面举例说明各个坐标轴上加速度值的计算方法，如图18-2所示。

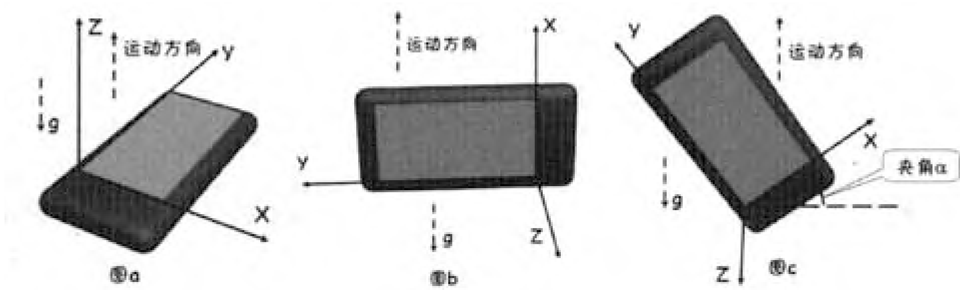


图18-2　几种情况下加速度值的计算

- 当手机屏幕与重力加速度方向垂直，以图a中的姿态以加速度a向上运动时，*Z* 轴的加速度值为（a+g）m/s^2 。这是因为此时手机本身在*Z* 轴的加速度为a，重力加速度方向沿*Z* 轴负方向，也就是重力加速度在*Z* 轴上的分量为-g，而“a-（-g）”的结果为a+g。
- 当手机的短边与重力加速度方向平行，以图b中的姿态以加速度a向上运动时，*X* 轴的加速度值为（a+g）m/s^2 。这是因为此时手机本身在*X* 轴的加速度为a，重力加速度方向沿*X* 轴负方向，也就是重力加速度在*X* 轴上的分量为-g，而“a-（-g）”的结果为a+g。

• 当手机屏幕与重力加速度方向平行，以图c中的姿态以加速度a向上运动时，*X* 轴的加速度值为（$a \cdot \sin\alpha + g \cdot \sin\alpha$）$m/s^2$ ，*Y* 轴的加速度值为（$a \cdot \cos\alpha + g \cdot \cos\alpha$）$m/s^2$ 。此情况下加速度的计算与前面两种情况类似，只是需要将加速度a与重力加速度g沿*X* 轴与*Y* 轴两个方向进行分解。

18.2.2　案例的开发

了解了加速度传感器的基本知识以后，就可以进行案例的开发了。由于上一节中在介绍传感器开发的基本流程时已经给出了本案例各部分的代码，因此这里仅给出案例的运行效果，如图18-3所示。



图18-3　案例运行效果图

说明

从图18-3中可以看出，随着手机与重力加速度方向之间的关系不同，随着手机加速度方向的不同，加速度在X、Y、Z坐标轴上的分量不断变化。

18.3　磁场传感器

本节主要介绍磁场传感器的使用，首先给出磁场传感器的一些基本知识，然后再给出一个简单的案例Sample18_2。

18.3.1 磁场传感器简介

磁场传感器主要用于探测手机周围的磁场强度，其与加速度传感器类似，也是返回磁场强度在X、Y、Z3个轴上的分量。返回数据的单位为uT，即微特斯拉。

另外，磁场传感器使用的坐标轴与加速度传感器的一样，*X* 轴平行于屏幕的短边，从左至右；*Y* 轴平行于屏幕的长边，从下至上；*Z* 轴垂直于屏幕与*X* 轴与*Y* 轴正交（见图18-1）。3个坐标轴也是绑定在手机上的，也就是说坐标轴与手机之间的关系不随着手机姿态的变化而变化。

> **提示**
>
> 由于每个轴磁场强度的计算方法与加速度传感器的基本一致，因此这里不再赘述。

18.3.2 案例的开发

了解了磁场传感器的基本知识后，就可以进行本节案例Sample18_2的开发了。进行代码的开发之前，请读者先了解一下本案例的运行效果，如图18-4所示。



图18-4　案例运行效果图-

说明

从图18-4中可以看出，随着手机姿态、朝向的变化，磁场在X、Y、Z坐标轴上的分量不断变化。同时，该案例还获取了手机中传感器的名称、类型、版本、耗电量等属性信息。

了解了案例的运行效果后，就可以进行代码的开发了。本案例中仅包含一个界面布局（main.xml）和一个Activity，布局的开发不是本章重点，这里不予介绍。下面仅给出本节案例中Activity的代码，具体内容如下。

代码位置：见随书光盘中源代码/第18章/Sample18_2/src/com/bn/Sample18_2目录下的MyActivity.java。

```java
package com.bn.Sample18_2;                            //包声明
……//此处省略部分类的引入代码，读者可以自行查看随书光盘中的源代码
public class MyActivity extends Activity {
    SensorManager mySensorManager;                    //SensorManager 对象引用
    Sensor myMagnetic;                                // Sensor 对象引用
    TextView tvX;                                     //显示磁场强度在 x 轴上分量的 TextView
    TextView tvY;                                     //显示磁场强度在 y 轴上分量的 TextView
    TextView tvZ;                                     //显示磁场强度在 z 轴上分量的 TextView
    TextView info;                                    //显示传感器属性信息的 TextView
      public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);                    //切换到主界面
        tvX = (TextView)findViewById(R.id.tvX);           //用于显示 x 轴方向磁场强度
        tvY = (TextView)findViewById(R.id.tvY);           //用于显示 y 轴方向磁场强度
        tvZ = (TextView)findViewById(R.id.tvZ);           //用于显示 z 轴方向磁场强度
        info= (TextView)findViewById(R.id.info);          //用于显示磁场传感器的属性信息
        //获得 SensorManager 对象
        mySensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
        //传感器的类型为磁场传感器
        myMagnetic=mySensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
        StringBuffer strb=new StringBuffer();      //创建一个 StringBuffer
        strb.append("\n 名称: ");
        strb.append(myMagnetic.getName());    //获取传感器名称
        strb.append("\n 耗电量(mA) : ");
        strb.append(myMagnetic.getPower());   //获取此传感器的耗电量，以毫安(mA)为单位
        strb.append("\n 类型编号  : ");
        strb.append(myMagnetic.getType());    //获取传感器类型编号
        strb.append("\n 制造商: ");
        strb.append(myMagnetic.getVendor()); //获取传感器的制造商
        strb.append("\n 版本: ");
        strb.append(myMagnetic.getVersion());//获取传感器版本
        strb.append("\n 最大测量范围: ");
        strb.append(myMagnetic.getMaximumRange());//获取传感器的最大测量范围(量程)
        info.setText(strb.toString());              //将信息放到显示用的 TextView 中
      }
      private SensorEventListener mySensorListener =
          new SensorEventListener(){//开发实现了 SensorEventListener 接口的监听器
          public void onAccuracyChanged(Sensor sensor, int accuracy){}
          public void onSensorChanged(SensorEvent event){
              float []values=event.values;//获取 X、Y、Z 3 个轴方向上的磁场强度分量值
              tvX.setText("x 轴方向上的磁场强度为:  "+values[0]);
              tvY.setText("y 轴方向上的磁场强度为:  "+values[1]);
              tvZ.setText("z 轴方向上的磁场强度为:  "+values[2]);
          }};
      protected void onResume(){/*此方法的实现与加速度传感器案例的实现基本相同，故省略*/}
      protected void onPause(){/*此方法的实现与加速度传感器案例的实现基本相同，故省略*/}
      public boolean onKeyDown(int keyCode,KeyEvent e){/*按下返回键退出，代码省略*/}
}
```

说明

从上述代码中可以看出，基本的套路是完全相同的。最大的变化就是在调用getDefaultSensor方法获取传感器对象时采用的参数由Sensor.TYPE_ACCELEROMETER变为了Sensor.TYPE_MAGNETIC_FIELD。

18.4　光传感器

本节主要介绍光传感器的使用，首先给出光传感器的一些基本知识，然后再给出一个简单的案例Sample18_3。

18.4.1　光传感器简介

光传感器主要用来探测手机所处环境的光照强度，此传感器仅有一个返回值，单位为勒克斯（lux）。灵活运用光传感器可以开发出非常人性化的应用程序。

设想如下的场景：为了应对不同光照强度的情况，应用程序的显示模式分为两种：白天与夜晚。当程序不够人性化时可能需要用户根据情况通过菜单、按钮等手动设置程序的工作模式。若采用了光传感器则可以根据光照的强度自动切换工作模式，这样就大大增加了程序对用户的吸引力。

18.4.2　案例的开发

了解了光传感器的基本知识后，就可以进行本节案例Sample18_3的开发了。进行代码的开发之前，请读者先了解一下本案例的运行效果，如图18-5所示。



图18-5　案例运行效果图

说明

从图18-5中可以看出，随着手机所处环境光照强度的不同，光传感器返回的数值也不断变化。同时，该案例也获取了手机中传感器的名称、类型、版本、耗电量等属性信息。

了解了本案例的运行效果后，就可以进行代码的开发了。由于本案例与本章前面的案例套路基本一致，只是细节有所不同。因此这里仅给出有特色的几处代码，具体如下所列。

（1）首先给出的是该案例中MyActivity类的onCreate方法，具体代码如下。

代码位置：见随书光盘中源代码/第18章/Sample18_3/src/com/bn/Sample18_3目录下MyActivity.java。

```
public void onCreate(Bundle savedInstanceState) {    //重写的 onCreate 方法
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);                   //切换到主界面
    light = (TextView)findViewById(R.id.light);      //用于显示光强度的 TextView
    info= (TextView)findViewById(R.id.info);     //用于显示光传感器属性信息的 TextView
    //获得 SensorManager 对象
    mySensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
    //获取传感器
    myLight=mySensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
        ……//省略了一些与前面案例相似的代码，读者可自行查阅随书光盘中的源代码
 }
```

说明

相比于前面的案例，主要的区别是在调用getDefaultSensor方法获取传感器对象时采用的参数变成了Sensor.TYPE_LIGHT。

（2）接着给出的是实现了SensorEventListener接口的传感器监听器，具体代码如下。

代码位置：见随书光盘中源代码/第18章/Sample18_3/src/com/bn/Sample18_3目录下MyActivity.java。

```
1 private SensorEventListener mySensorListener =
2      new SensorEventListener(){           //实现了SensorEventListener接口的传感器监听器
3      @Override                            //重写onAccuracyChanged方法
4      public void onAccuracyChanged(Sensor sensor, int accuracy){}
5      @Override                            //重写onSensorChanged方法
6      public void onSensorChanged(SensorEvent event){
7          float []values=event.values;     //获取光照强度值所属的values数组
8          light.setText("光的强度为："+values[0]); //显示光照强度
9      }};
```

说明

从上述代码的第7-8行中可以看出，与前面的加速度传感器与磁场传感器不同，光传感器仅有一个返回值，就是光照强度。

18.5　温度传感器

本节主要介绍温度传感器的使用，首先给出温度传感器的一些基本知识，然后再给出一个简单的案例Sample18_4。

18.5.1　温度传感器简介

温度传感器主要用来探测手机所处环境的温度，此传感器仅有一个返回值，单位为摄氏度。灵活运用温度传感器可以开发出非常有意

思的应用程序，如温度计、与温度相关的健康指南、饮食指南等。

提示

上述笔者仅仅是抛砖引玉，读者可以充分发挥想象，创造出各种各样有价值的应用程序。

18.5.2　案例的开发

了解了温度传感器的基本知识后，就可以进行本节案例Sample18_4的开发了。进行代码的开发之前，请读者先了解一下本案例的运行效果，如图18-6所示。



图18-6　案例运行效果图

说明

从图18-6中可以看出随手机所处环境温度的不同，获取的相应温度值不断变化。同时，该案例也获取了手机中传感器的名称、类型、版本、耗电量等属性信息。

了解了本案例的运行效果后，就可以进行代码的开发了。由于本案例与本章前面的案例套路基本一致，只是细节有所不同。因此这里仅给出有特色的几处代码，具体如下所列。

（1）首先给出的是该案例中MyActivity类的onCreate方法，具体代码如下。

代码位置：见随书光盘中源代码/第18章/Sample18_4/src/com/bn/Sample18_4目录下MyActivity.java。

```
public void onCreate(Bundle savedInstanceState) {          //重写的 onCreate 方法
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);                          //切换到主界面
    temperature = (TextView)findViewById(R.id.temperature);   //用于显示温度的 TextView
    info= (TextView)findViewById(R.id.info);//用于显示温度传感器属性信息的 TextView
    //获得 SensorManager 对象
    mySensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
    //获取传感器
    myTemperature=mySensorManager.getDefaultSensor(Sensor.TYPE_TEMPERATURE);
        ……//省略了一些与前面案例相似的代码，读者可自行查阅随书光盘中的源代码
}
```

> **说明**
>
> 相比于前面的案例，主要的区别是在调用getDefaultSensor方法获取传感器对象时采用的参数变成了Sensor.TYPE_TEMPERATURE。

（2）接着给出的是实现了SensorEventListener接口的传感器监听器，具体代码如下。

代码位置：见随书光盘中源代码/第18章/Sample18_4/src/com/bn/Sample18_4目录下MyActivity.java。

```
private SensorEventListener mySensorListener =
    new SensorEventListener(){          //实现了 SensorEventListener 接口的传感器监听器
    @Override                           //重写 onAccuracyChanged 方法
    public void onAccuracyChanged(Sensor sensor, int accuracy){}
    @Override                           //重写 onSensorChanged 方法
    public void onSensorChanged(SensorEvent event){
        float []values=event.values;        //获取温度值所属的 values 数组
        temperature.setText("温度为: "+values[0]);          //显示温度
    }};
```

说明

从上述代码的第7-8行中可以看出，与前面的光传感器相同，温度传感器仅有一个返回值，就是温度值（以摄氏度衡量）。

18.6 接近传感器

本节主要介绍接近传感器的使用，首先给出接近传感器的一些基本知识，然后再给出一个简单的案例Sample18_5。

18.6.1 接近传感器简介

接近传感器用于探测是否有物体离手机屏幕非常近（在1厘米左右的范围内），此传感器仅有一个返回值，表示物体距手机屏幕的距离。返回值随手机型号的不同会有所差异，但无论哪种型号的手机都只有两种可能，一种表示有物体在离手机屏幕近的范围内，另一种表示不在范围内。

如笔者使用的摩托罗拉里程碑2用2.1474837E8表示不在近的范围内，用3表示在近的范围内。而有些型号的手机用100表示不在近的范围内，用0表示在近的范围内。

恰当使用接近传感器可以开发出很人性化的应用程序，如手机上的通话程序就是如此。当拿着手机贴到耳朵上通话时，接近传感器检测到耳朵离屏幕很近，则关闭屏幕以节约用电；当结束通话，手机屏幕离开耳朵时，接近传感器检测到没有物体离屏幕很近了，则自动打开屏幕。

提示

有些资料中将此传感器错误地解释为距离传感器是不完全准确的，因为此传感器不能用于测量连续的距离值，仅可以返回两个离散值代表接近与非接近两种状态。另外，此传感器一般位于屏幕的左上侧。

18.6.2 案例的开发

了解了接近传感器的基本知识后，就可以进行本节案例Sample18_5的开发了。进行代码的开发之前，请读者先了解一下本案例的运行效果，如图18-7所示。



图18-7　案例运行效果图

说明

从图18-7中可以看出随物体离手机距离的不同，接近传感器的返回值在两种之间变化。同时，该案例也获取了手机中传感器的名称、类型、版本、耗电量等属性信息。

了解了本案例的运行效果后，就可以进行代码的开发了。由于本案例与本章前面的案例套路基本一致，只是细节有所不同。因此这里仅给出有特色的几处代码，具体如下所列。

（1）首先给出的是该案例中MyActivity类的onCreate方法，具体代码如下。

代码位置：见随书光盘中源代码/第18章/Sample18_3/src/com/bn/Sample18_5目录下MyActivity.java。

```
public void onCreate(Bundle savedInstanceState) {        //重写的 onCreate 方法
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);                        //切换到主界面
    distance = (TextView)findViewById(R.id.distance);     //用于显示距离的 TextView
    info= (TextView)findViewById(R.id.info);//用于显示手机中接近传感器的相关信息
    //获得 SensorManager 对象
    mySensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
    //获取传感器
    myAccelerometer=mySensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY);
        ……//省略一些与前面案例相似的代码，读者可自行查阅随书光盘中的源代码
  }
```

说明

相比于前面的案例，主要的区别是在调用getDefaultSensor方法获取传感器对象时采用的参数变成了Sensor.TYPE_PROXIMITY。

（2）接着给出的是实现了SensorEventListener接口的传感器监听器，具体代码如下。

代码位置：见随书光盘中源代码/第18章/Sample18_3/src/com/bn/Sample18_5目录下MyActivity.java。

```
1 private SensorEventListener mySensorListener =
2     new SensorEventListener(){           //实现了 SensorEventListener 接口的传感器监听器
3     @Override                            //重写 onAccuracyChanged 方法
4     public void onAccuracyChanged(Sensor sensor, int accuracy){}
5     @Override                            //重写 onSensorChanged 方法
6     public void onSensorChanged(SensorEvent event){
7         float []values=event.values;     //获取距离值所属的 values 数组
8         distance.setText("距离为: "+values[0]);            //显示距离
9     }};
```

说明

从上述代码的第7-8行中可以看出，与前面的温度传感器、光传感器相同，接近传感器仅有一个返回值，那就是距离。不过正像前面特别强调的那样，这里的距离仅仅是两个离散值，一个表示接近状态，另一个表示非接近状态。

18.7 姿态传感器

本节主要介绍姿态传感器的使用，首先给出姿态传感器的一些基本知识，然后再给出一个简单的案例Sample18_6。

18.7.1 姿态传感器简介

本章介绍的所有传感器中，姿态传感器是与游戏开发人员关系最为密切的一个了。很多的智能手机中的体感游戏，如都市赛车、极品飞车等，都是采用姿态传感器进行操控的。具体说就是玩家在游戏过程中，根据操控的需要改变手机的姿态，姿态传感器获得姿态数据后传递给应用程序进行分析、计算，得出车辆等被操控物体的运动情况。

从上述介绍中可以看出，姿态传感器主要用于感知手机姿态的变化，其每次读取的都是静态的状态值，表示当前的姿态。每组姿态值包括3个，分别表示手机在Yaw、Pitch、Roll轴的旋转角度。Yaw、Pitch、Roll轴与手机之间的关系比较复杂，详细情况如图18-8所示。

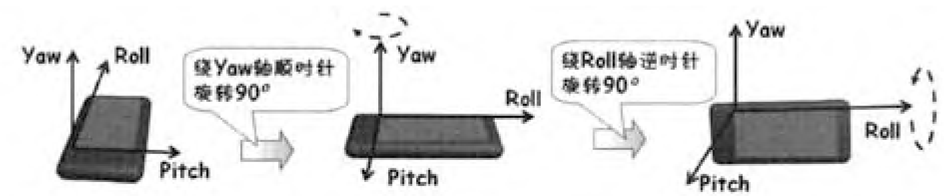


图18-8　手机姿态传感器

说明

图18-8中从左至右分别为手机在原始姿态，手机绕Yaw轴顺时针旋转90。后的姿态，手机再绕Roll轴逆时针旋转90。后的姿态。从几幅图的对比中可以看出，Yaw、Pitch、Roll 3个轴与手机之间的关系不都是固定的。

Yaw、Pitch、Roll 3个轴的详细情况如下所列。

- Yaw轴。

Yaw从SDK的2.3版本后改称为Azimuth轴了，无论手机处于何种姿态，此轴都与重力加速度方向相反，垂直向上。也就是说，此轴是固定不变的。因此姿态传感器工作时，获得的此轴角度表示的是手机的方位。

具体情况为：0°时手机指向北方，90°时指向东方，180°时指向南方，270°时指向西方，其他的方位可以依此类推。因此，在原始状态时手机屏幕水平向上（即屏幕的法向量与重力加速度方向相反），指向北方。

- Pitch轴。

Pitch轴的方向与当前手机的方位角密切相关，原始情况下Pitch轴指向东方。当手机绕Yaw轴旋转一定的角度后，Pitch轴也绕Yaw轴旋转相同的角度。如图18-8中间的小图所示，当手机绕Yaw轴顺时针旋转90°后，Pitch轴也绕Yaw轴顺时针旋转90°，指向南方。

简单来说，可以将Pitch轴理解为焊死在Yaw上的。另外，要注意的是，Pitch轴与手机之间的关系是不固定的。如图18-8中右侧的小图所示，当手机又绕Roll轴旋转后，Pitch轴与手机之间的关系就发生了变化。

- Roll轴。

从图18-8中可以看出，Roll轴的方向依赖于手机绕Yaw轴和Pitch轴旋转的情况，但是Roll轴的确定并不复杂。细心观察一下图18-8即可发现，Roll轴与手机之间的关系是固定的，就像是焊死在手机上一样。

看完上面的介绍后，读者可能会有这样的疑问：为什么3个轴这么古怪呢？其实不然，这3个轴是来自数学上的欧拉角。欧拉角是一种表示物体姿态的方法，在飞机飞行中很常用，具体情况如图18-9所示。

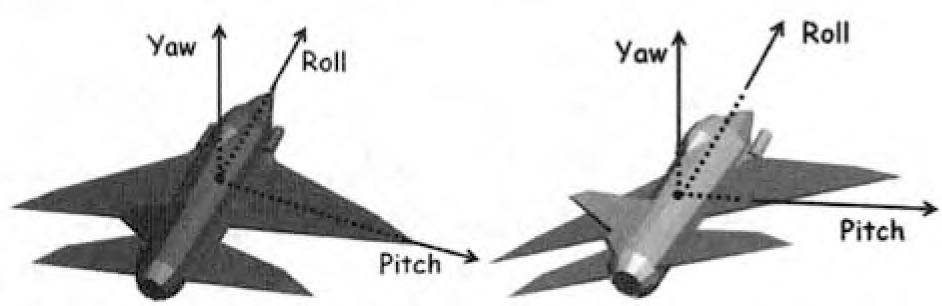


图18-9　用欧拉角表示飞机姿态

从图18-9中可以看出，Yaw轴角度为飞机的方位角，Pitch轴角度为飞机的俯仰角，Roll轴角度为飞机的左右倾斜角。知道了这3个角度值后，飞机的姿态就唯一地被确定了。上述说法是从飞行员的角度出发的，乘客也是如此，坐过飞机的读者可以很容易地理解。

说明

关于欧拉角的相关数学问题的讨论已经大大超过了本书既定的覆盖范围，有需要的读者可以自行查阅相关的数学资料。

18.7.2 案例的开发

通过上一小节的学习，读者对姿态传感器的3个轴应该有所了解了。但如何通过3个轴的采样 值进行计算、分析得出游戏中被操控物体的运动方向还不是很清楚。本节将通过一个案例（Sample18_6）向读者介绍如何在实际开发中通过姿态传感器操控物体，其运行效果如图18-10所示。

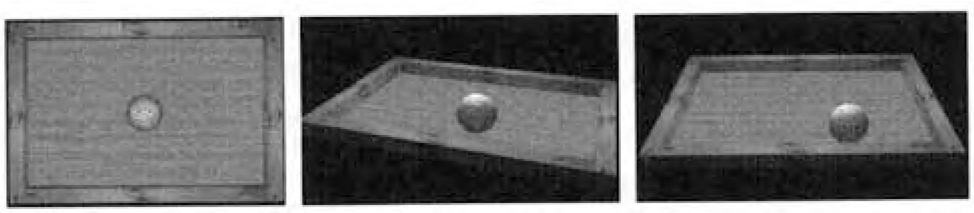


图18-10 案例运行效果图

说明

从图18-10中可以看出，本案例呈现了一个3D场景。场景中有一个四周凸出的矩形木头盘，其中放置了一个空心圆球。当手机倾斜时，圆球会向低处滚动。另外，程序运行时用手指在屏幕上滑动可以改变摄像机的位置，图18-10中的3幅小图就是摄像机从不同位置观察场景时的情况。

了解了案例的运行效果后，下面简要介绍一下本案例中是如何将Yaw、Pitch、Roll 3个轴的读数正确转化为小球的运动方向的，基本思路如图18-11所示。

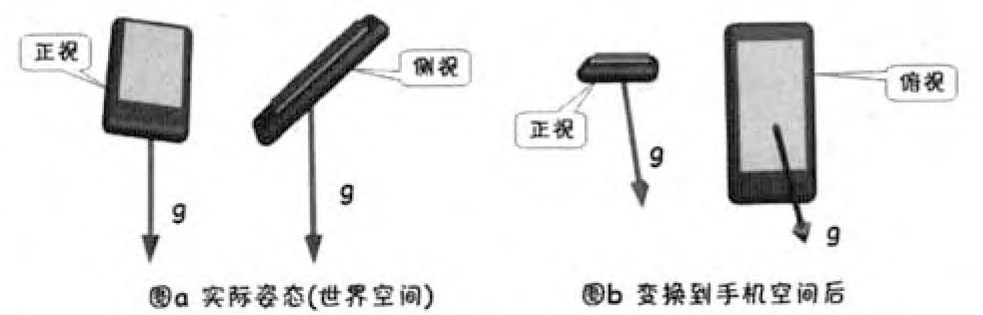


图18-11　计算思路示意图

> **说明**
>
> 图18-11中左右两侧的图a和图b分别为某时刻实际的手机姿态与重力加速度的方向以及转换到手机空间后手机的姿态以及重力加速度的方向。另外，图a与图b中各自都有两幅小图，是分别从不同的角度观察的结果，也就是说两幅小图对应的手机姿态是相同的。

将Yaw、Pitch、Roll 3个轴的读数正确转化为小球运动方向的思路也很简单：屏幕上的小球是向下滚动的，因此应该沿重力加速度向量在屏幕上投影向量的方向运动，具体情况如下所列。

- 从图a中可以看出，在实际空间中重力加速度垂直向下，方向很容易确定。但此时手机屏幕平面的姿态是通过欧拉角3个轴的读数确定的，要想将重力加速度直接投影到屏幕平面上计算是非常复杂的。
- 图b是将图a中的重力加速度向量焊死到手机上，然后通过欧拉角3个轴的读数将手机绕Yaw、Pitch、Roll轴反转恢复得到的情况。此时手机恢复到了原始状态，屏幕平面水平向上。同时由于旋转恢复时是将重力加速度焊死在手机上的，因此旋转后重力加速度向量与手机之间的关系不变。
- 因此求图a中垂直向下的重力加速度向量在任意姿态屏幕平面上的投影与求图b中旋转恢复 后重力加速度向量在水平屏幕平面上的投影结果是相同的。但计算复杂度就相差甚远了，图b中求投影时仅需舍弃重力加速度向量与屏幕平面垂直的分量即可。

提示

从图a到图b的变换实际上就是从世界空间到手机空间的变换。所谓世界空间是指实际情况所对应的空间坐标系，手机空间是指以手机为中心，认为手机处于标准原始姿态的坐标系。3D应用程序的开发中，进行坐标系转换是一种常用的技术手段，可以大大地将问题简化。

了解了案例的运行效果与计算思路后，就可以进行代码的开发了。由于本案例中的大部分代码来自于前面章节的不同案例，因此这里对重复的内容不再赘述，仅给出该案例中有特色的类以及方法。

（1）首先给出的是用于将重力加速度向量根据欧拉角三轴读数转换到手机原始空间的工具类RotateUtil，其代码如下。

代码位置：见随书光盘中源代码/第18

章/Sample18_6/src/com/bn/Sample18_6目录下RotateUtil.java。

```
package com.bn.Sample18_6;                                   //包声明
public class RotateUtil{
    public static double[] pitchRotate(double angle,double[] gVector){
        double[][] matrix={                                  //绕 Pitch 轴旋转的变换矩阵
          {1,0,0,0},                        //该矩阵的作用为将向量绕 Pitch 轴旋转 angle 度
          {0,Math.cos(angle),Math.sin(angle),0},
          {0,-Math.sin(angle),Math.cos(angle),0},
          {0,0,0,1}};
        double[] tempDot={gVector[0],gVector[1],gVector[2],gVector[3]};//复制重力加速度向量
            for(int j=0;j<4;j++){                            //重力加速度向量与矩阵相乘
                gVector[j]=(tempDot[0]*matrix[0][j]+tempDot[1]*matrix[1][j]+
                        tempDot[2]*matrix[2][j]+tempDot[3]*matrix[3][j]);
            }
            return gVector;                                  //返回绕 Pitch 轴旋转后的向量
        }
        public static double[] rollRotate(double angle,double[] gVector){
            double[][] matrix={                              //绕 Roll 轴旋转的变换矩阵
              {Math.cos(angle),0,-Math.sin(angle),0},
              {0,1,0,0},                    //该矩阵的作用为将向量绕 Roll 轴旋转 angle 度
              {Math.sin(angle),0,Math.cos(angle),0},
              {0,0,0,1} };
            double[] tempDot={gVector[0],gVector[1],gVector[2],gVector[3]};
                                                             //复制重力加速度向量
            for(int j=0;j<4;j++){                            //重力加速度向量与矩阵相乘
                gVector[j]=(tempDot[0]*matrix[0][j]+tempDot[1]*matrix[1][j]+
                          tempDot[2]*matrix[2][j]+tempDot[3]*matrix[3][j]);
            }
            return gVector;                                  //返回绕 Roll 轴旋转后的向量
        }
        public static double[] yawRotate(double angle,double[] gVector){
            double[][] matrix={                              //绕 Yaw 轴旋转的变换矩阵
              {Math.cos(angle),Math.sin(angle),0,0},
              {-Math.sin(angle),Math.cos(angle),0,0},
```

```
33            {0,0,1,0},                    //该矩阵的作用为将向量绕 Yaw 轴旋转 angle 度
34            {0,0,0,1}
35          };
36          double[] tempDot={gVector[0],gVector[1],gVector[2],gVector[3]};
                                            //复制重力加速度向量
37          for(int j=0;j<4;j++){           //重力加速度向量与矩阵相乘
38              gVector[j]=(tempDot[0]*matrix[0][j]+tempDot[1]*matrix[1][j]+
39                          tempDot[2]*matrix[2][j]+tempDot[3]*matrix[3][j]);
40          }
41          return gVector;                 //返回绕 Yaw 轴旋转后的向量
42      }
43      public static float[] getDirectionDot(double[] values){
44          double yawAngle=-Math.toRadians(values[0]);      //获取 Yaw 轴旋转角度弧度
45          double pitchAngle=-Math.toRadians(values[1]);    //获取 Pitch 轴旋转角度弧度
46          double rollAngle=-Math.toRadians(values[2]);     //获取 Roll 轴旋转角度弧度
47          double[] gVector={0,0,-100,1};                //虚拟一个重力向量(垂直向下)
48          gVector=RotateUtil.yawRotate(yawAngle,gVector); //绕 Yaw 轴旋转恢复
49          gVector=RotateUtil.pitchRotate(pitchAngle,gVector); //绕 Pitch 轴旋转恢复
50          gVector=RotateUtil.rollRotate(rollAngle,gVector);   //绕 Roll 轴旋转恢复
51          double mapX=gVector[0];                       //变换后重力加速度向量的 X 分量
52          double mapY=gVector[1];                       //变换后重力加速度向量的 Y 分量
53          double mapZ=gVector[2];                       //变换后重力加速度向量的 Z 分量
54          float[] result={(float) mapX,(float) mapY,(float) mapZ};
55          return result;                  //返回变换到手机空间的重力加速度向量
56  }}
```

- 第3-42行为绕Yaw、Pitch、Roll轴进行恢复旋转的工具方法，旋转计算采用的是矩阵实现。若读者对通过采用矩阵进行旋转变换的问题不是很了解，可以参考计算机图形学的相关资料，这些内容超出了本书的既定范围，将不进行讨论。
- 第43-56行为实现前面介绍的计算思路的getDirectionDot方法，其将重力加速度向量依次绕Yaw、Pitch、Roll轴进行恢复旋转，以变换到手机空间轻松得到重力加速度向量在手机屏幕平面的投影。

（2）接着给出的是案例的主控制类——MyActivity，其代码如下。

代码位置：见随书光盘中源代码/第18章/Sample18_6/src/com/bn/Sample18_6目录下MyActivity.java。

```
package com.bn.Sample18_6;                                    //包声明
......//此处省略部分相关类的引入代码，读者可自行查看随书光盘的源代码
public class MyActivity extends Activity {
    SensorManager mySensorManager;                             //SensorManager 类的引用
    Sensor mySensor;                                           //姿态传感器的引用
    MySurfaceView mySurfaceView;
   public void onCreate(Bundle savedInstanceState) {           //重写 onCreate 方法
      super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE); //设置为无标题
            getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN ,
             WindowManager.LayoutParams.FLAG_FULLSCREEN);//设置为全屏
            //设置为横屏模式
            setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
          //获得 SensorManager
          mySensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
          //获取姿态传感器对象
          mySensor=mySensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION);
          mySurfaceView = new MySurfaceView(this); //创建呈现场景的 MySurfaceView 对象
          this.setContentView(mySurfaceView);       //切换到场景界面
          mySurfaceView.requestFocus();             //获取焦点
          mySurfaceView.setFocusableInTouchMode(true);  //设置为可触控
      }
      //实现了 SensorEventListener 接口的传感器监听器
      private SensorEventListener mySensorListener=new SensorEventListener(){
          public void onAccuracyChanged(Sensor sensor, int accuracy){}
          public void onSensorChanged(SensorEvent event){ //重写 onSensorChanged 方法
            float []values=event.values;            //获取 3 个轴的旋转角度值
            int directionDotXYZ[]=RotateUtil.getDirectionDot(//求重力加速度向量投影
               new double[]{values[0],values[1],values[2]}
            );
            double mLength=directionDotXYZ[0]*directionDotXYZ[0]+//求出投影后向量模的平方
                        directionDotXYZ[1]*directionDotXYZ[1];
            mLength=Math.sqrt(mLength);             //求出投影后向量的模
            if(mLength==0){return;}                 //若 mLength 为零，则不执行下面的代码
            if( directionDotXYZ[2]<0){//若屏幕是朝上的则用下列代码计算球的位移
                  Constant.SPANX=(float)((directionDotXYZ[1]/mLength)*0.08f);
                  Constant.SPANZ=(float)((directionDotXYZ[0]/mLength)*0.08f);
              }else{//若屏幕是朝下的则用下列代码计算球的位移
                  Constant.SPANX=(float)((directionDotXYZ[1]/mLength)*0.08f);
                  Constant.SPANZ=-(float)((directionDotXYZ[0]/mLength)*0.08f);
              }
      }};
      protected void onResume() {/*省略该方法的内容*/}
      protected void onPause() {/*省略该方法的内容*/}
      public boolean onKeyDown(int keyCode,KeyEvent e){/*省略该方法的内容*/}
  }
```

提示

上述代码中第24-42行是本案例的关键点之一，为传感器监听器的实现。在此传感器监听器的onSensorChanged方法中，程序首先将从姿态传感器获取的欧拉角三个轴的值传给工具方法getDirectionDot，getDirectionDot方法根据接收的角度值将重力加速度向量进行旋转变换到手机空间中，最后根据变换后重力加速度向量的3个分量计算出小球的位移。

（3）最后给出的是用于描述小球位置姿态的BallForControl类，其代码如下。

代码位置：见随书光盘中源代码/第18章/Sample18_6/src/com/bn/Sample18_6目录下BallForControl.java。

```
package com.bn.Sample18_6;                              //包声明
……//此处省略部分相关类的引入代码，读者可自行查看随书光盘的源代码
public class BallForControl {
    MySurfaceView mv;                                   // MySurfaceView类的引用
    Ball ball;                                          //用于绘制的球(采用的是第8章的线框几何球)
    ……//此处省略相关成员变量的声明，读者可自行查看随书光盘的源代码
```

```
8    float[] selfRotateMatrix;                     //自带的记录当前姿态的旋转矩阵
9    public BallForControl(MySurfaceView mv,float scale,float aHalf,int n){
10           this.mv=mv;
11           ball=new Ball(mv,scale,aHalf,n);   //创建用于绘制的几何球对象
12           selfRotateMatrix=new float[16];    //初始化自带的记录当前姿态的旋转矩阵
13           Matrix.setRotateM(selfRotateMatrix, 0, 10, 0, 1, 0);//初始时旋转一定的度数
14       }
15       public void drawSelf(){                   //绘制球的 drawSelf 方法
16           MatrixState.pushMatrix();             //保护现场
17           MatrixState.translate(Constant.XOFFSET, 1, Constant.ZOFFSET);//移动到指定位置
18           MatrixState.matrix(selfRotateMatrix);  //加上记录姿态的旋转矩阵
19           ball.drawSelf(ballTexId);              //绘制球
20           MatrixState.popMatrix();               //恢复现场
21       }
22       public void go(){                          //球按照当前重力方向移动的方法
23           tempSPANX=Constant.SPANX;              //球移动距离的临时变量赋值
24           tempSPANZ=Constant.SPANZ;
25           tempX=Constant.XOFFSET+tempSPANX;//根据传感器计算结果，改变当前球的位置
26           tempZ=Constant.ZOFFSET+tempSPANZ;
27           if( (tempZ<-ZBOUNDARY)||(tempZ>ZBOUNDARY)){//如果与上下两条边发生碰撞
28               tempSPANZ=0;                            //z 轴上步进为 0
29           }
30           if((tempX<-XBOUNDARY)|| (tempX>XBOUNDARY)){//如果与左右两条边发生碰撞
31               tempSPANX=0;                       //x 轴上步进为 0
32           }
33           Constant.XOFFSET+=tempSPANX;           //球当前的位置事实发生变化
34           Constant.ZOFFSET+=tempSPANZ;
35           rotateX=tempSPANZ;                     //计算旋转轴向量的 x、y、z 分量
36           rotateY=0;
37           rotateZ=-tempSPANX;
38           tempLength=(float) Math.sqrt           //计算球前进的距离
39               (tempSPANX*tempSPANX+tempSPANZ*tempSPANZ);
40           tempAngle=(float) Math.toDegrees       //计算前进的角度值
41               (tempLength/Constant.BALLR);
42           if(Math.abs(tempAngle)!=0&&         //若球滚动了则改变球的旋转矩阵
43                   (Math.abs(rotateZ)!=0||Math.abs(rotateX)!=0)){
44               float[] newMatrix=new float[16];   //新建旋转矩阵
45               Matrix.setRotateM(newMatrix, 0, tempAngle, rotateX, rotateY, rotateZ);
46               float[] resultMatrix=new float[16];//将新旋转矩阵与原来的旋转矩阵综合
47               Matrix.multiplyMM(resultMatrix, 0, newMatrix, 0, selfRotateMatrix,0);
48               selfRotateMatrix=resultMatrix;     //返回最终旋转矩阵
49   }}}
```

说明

上述代码中最重要的是第22-49行的go方法，其根据姿态传感器监听器的计算结果将小球的位置姿态进行恰当的变换供绘制时使用。要注意的是，由于OpenGL ES 2.0是需要开发人员向渲染管线提供矩阵的，因此出于方便考虑，本案例中小球的姿态记录在了一个旋转矩阵中。可以记录姿态的还有四元数，有兴趣的读者可以自行实现。

18.8　本章小结

本章详细地向读者介绍了Android平台下6种不同传感器的使用，其中与3D游戏开发关系最为密切的就是姿态传感器。因此本章也针对姿态传感器给出了一个非常完整的案例，掌握此案例后，读者在以后3D游戏的开发中可以轻松应对各种体感操作的需求。

第19章

游戏开发小贴士

本章将介绍几种在3D游戏开发中常用的技术，主要包括：3D拾取、多点触控、多键监听等。掌握了这几种技术以后，读者应该可以开发出更好的人机交互界面，大大提高游戏或应用的吸引力。

19.1 3D拾取技术

很多3D游戏中都允许用户通过触摸屏幕选中虚拟3D世界中的物体以进行操控，这时就需要用到3D拾取技术。其是3D游戏开发中必知必会的技术之一，本节将对其进行详细介绍。

19.1.1　案例效果与基本原理

本节主要是通过一个使用了3D拾取技术的案例Sample19_1对其进行介绍，在介绍本节案例的具体开发之前首先需要了解一下其运行效果，如图19-1所示。

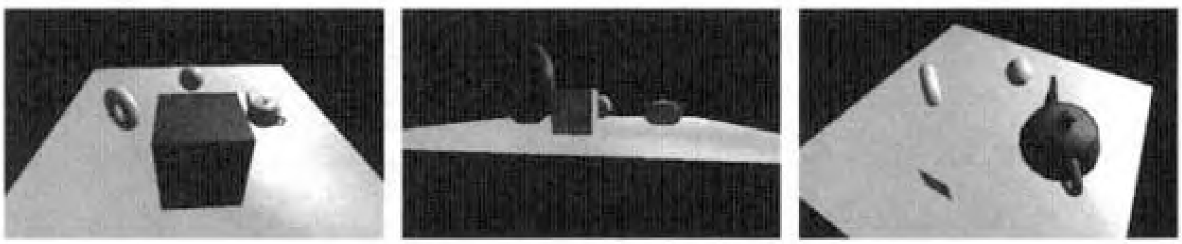


图19-1　案例Sample19_1的运行效果

提示

运行本案例时，当手指在屏幕上水平滑动时摄像机会绕场景转动，当手指在屏幕上垂直滑动时摄像机会随手指的移动而升降。直接在屏幕上单击物体就可以使被选中的物体改变颜色和大小，也就是选中的物体会变大，并且从灰白色变为绿色。由于插图是灰度印刷，颜色的变化可能看得不是很清楚，此时请读者自行采用真机运行本案例即可。

了解了本节案例的运行效果后，下面应该了解的是3D拾取技术的基本原理，如图19-2所示。

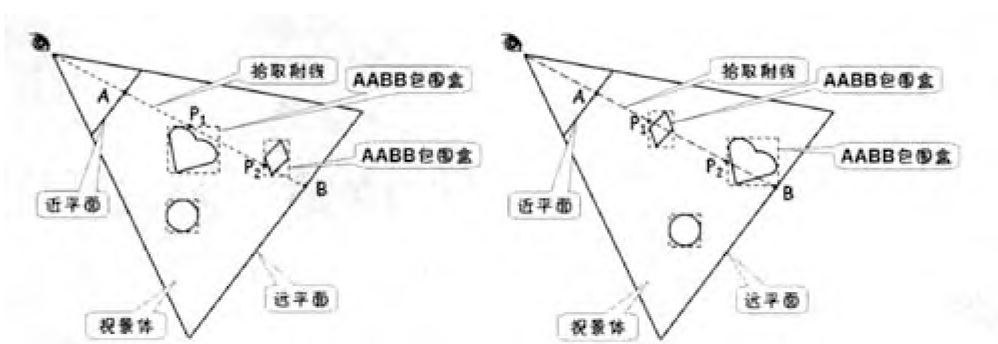


图19-2　3D拾取技术原理

从图19-2中可以看出，3D拾取的基本思想非常简单。由摄像机与屏幕上的触控点确定一条拾取射线，求出拾取射线与场景中物体AABB包围盒最近的交点，此交点所属AABB包围盒对应的物体即为被选中的物体。

提示

从图19-2中可以看出，由于是基于物体的AABB包围盒进行的拾取计算，使得计算并不是“绝对”精确的。如左图中，若精确计算选中的应该是远处的物体，但基于AABB包围盒的计算结果选中的则是近处的心形物体。虽然基于AABB包围盒进行计算会有一些误差，但在大部分情况下并不明显，而且基于AABB包围盒进行计算效率会高很多，因此大部分情况下还是采用此方案。

由于在OpenGL ES 2.0中只有位于视景体里的物体才可能被看见，因此问题可以转化为求线段AB与场景中物体AABB包围盒离A点最近的交点，此交点所属AABB包围盒对应的物体即为被选中的物体。从上面的讨论中可以看出，关键的就是如何通过屏幕上的触控点位置求出A、B两点的坐标了，具体步骤如下。

- 虽然最终希望求出的是A、B两点在3D虚拟空间（世界坐标系）中的坐标，但为了易于开发，首先应该求出A、B两点在摄像机坐标系中的三维坐标。最先求出的是近平面上A点的坐标，如图19-3所示。

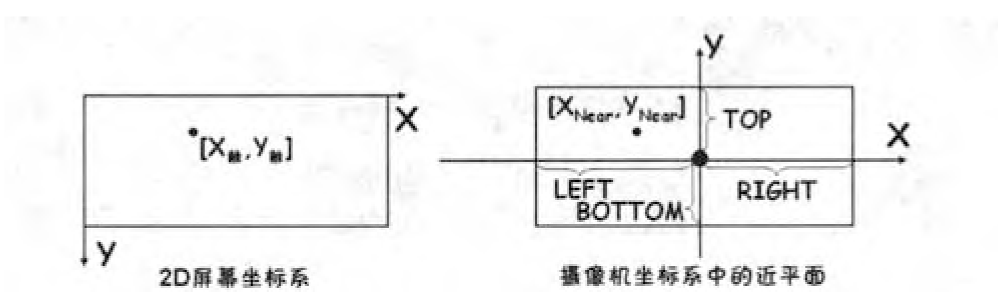


图19-3　将2D屏幕坐标系中触控点坐标换算为摄像机坐标系近平面坐标

从图19-3中可以看出，触控点的坐标与摄像机近平面上的坐标是线性相关的，转换公式如下。

X_{Near} =（$X_{触}$ -HW）*LLR/HW　Y_{Near} =（HH-$Y_{触}$ ）*LTB/HH

其中HW、LLR、HH、LTB的含义如下。

LEFT=RIGHT=LLR TOP=BOTTOM=LTB

SCREEN_WIDTH/2=HW SCREEN_HEIGHT/2=HH

另外，由于在摄像机坐标系中认为摄像机位于原点，沿Z轴负方向观察，因此换算后近平面上A点的Z坐标为-near，其中near为近平面与摄像机之间的距离。综上，求得的A点坐标如下。

$$[（X_{触} -HW）*LLR/HW,（HH-Y_{触}）*LTB/HH，-near]$$

• 求出近平面上A点的坐标后，根据相似三角形的原理即可求出远平面上B点的坐标，如图 19-4所示。

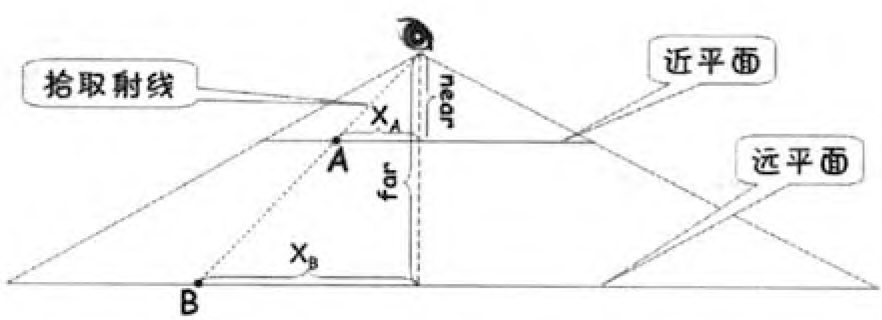


图19-4 通过A点坐标求B点坐标

从图19-4中可以看出，根据相似三角形的原理，X_A 、X_B 、near, far之间满足如下等式。

$$X_A /X_B =near/far$$

同理，Y_A 、Y_B 、near、far之间满足如下等式。

$$Y_A /Y_B =near/far$$

综上，求得的B点坐标如下。

$$[far•X_A /near, far•Y_A /near,-far]$$

• 求得A、B两点在摄像机坐标系中的坐标之后，再将此坐标经摄像机矩阵的逆矩阵变换即可得到A、B两点在世界坐标系中的坐标。

顺利求得A、B两点在世界坐标系中的坐标后，就可以轻松得出线段AB的解析方程，并计算出距离A点最近的交点以求出当前被选中的物体了。

提示

求最近交点的方法与前面第15章中光线跟踪采用的方法类似，这里就不再赘述了，需要的读者请参考第15章中介绍光线跟踪的相关部分。

19.1.2 开发步骤

上一小节介绍了3D拾取技术的基本原理及案例的运行效果，本小节将介绍如何进行本节案例的开发。由于本节案例中很多类与前面不少案例中的非常类似，因此这里仅介绍与3D拾取密切相关的部分，具体内容如下。

（1）首先在MatrixState类中增加求摄像机矩阵逆矩阵的getlnvertMvMatrix方法以及将摄像机坐标系中坐标变换为世界坐标系中坐标的fromPtoPreP方法，具体代码如下。

代码位置：见随书光盘中源代码/第19章/Sample19_1/com/bn/Sample19_1目录下的MatrixState.java。

```java
public static float[] getInvertMvMatrix(){       //获取摄像机矩阵的逆矩阵的方法
    float[] invM = new float[16];
    Matrix.invertM(invM, 0, mVMatrix, 0);        //求摄像机矩阵的逆矩阵
    return invM;
}
public static float[] fromPtoPreP(float[] p){ //将摄像机坐标系中坐标变换为世界坐标系中坐标
    float[] inverM = getInvertMvMatrix();        //获取摄像机矩阵的逆矩阵
    float[] preP = new float[4];                 //用于存储结果坐标的数组
    Matrix.multiplyMV(preP, 0, inverM, 0, new float[]{p[0],p[1],p[2],1}, 0);
                                                 //求世界坐标系中的坐标
        return new float[]{preP[0],preP[1],preP[2]};   //返回世界坐标系中的坐标
    }
```

（2）接下来开发计算AB两点坐标的工具类IntersectantUtil，其代码如下。

代码位置：见随书光盘中源代码/第19章/Sample19_1/com/bn/Sample19_1目录下的IntersectantUtil.java。

```
package com.bn.Sample19_1;                    //声明包
public class IntersectantUtil {
    public static float[] calculateABPosition(          //计算AB点坐标的方法
        float x, float y,                    //触控点XY坐标
        float w, float h,                    //屏幕宽度、高度
        float left, float top, float near, float far){   //近平面参数
        float x0=x-w/2;                      //将以屏幕左上角为原点的触控点坐标换算
        float y0=h/2-y;                      //为以屏幕中心为坐标原点的坐标
        float xNear=2*x0*left/w;             //计算对应的near面上的x、y坐标
            float yNear=2*y0*top/h;
            float ratio=far/near;            //计算far与near的比值
            float xFar=ratio*xNear;          //计算对应的far面上的x、y坐标
            float yFar=ratio*yNear;
          float ax=xNear;                    //摄像机坐标系中A点的X坐标
          float ay=yNear;                    //摄像机坐标系中A点的Y坐标
          float az=-near;                    //摄像机坐标系中A点的Z坐标
          float bx=xFar;                     //摄像机坐标系中B点的X坐标
          float by=yFar;                     //摄像机坐标系中B点的Y坐标
          float bz=-far;                     //摄像机坐标系中B点的Z坐标
           //通过摄像机坐标系中A、B两点的坐标以及摄像机矩阵的逆矩阵,
           //求世界坐标系中A、B两点的坐标
           float[] A = MatrixState.fromPtoPreP(new float[] { ax, ay, az });
                                             //世界坐标系中A点坐标
           float[] B = MatrixState.fromPtoPreP(new float[] { bx, by, bz });
                                             //世界坐标系中B点坐标
           return new float[] {              //返回AB两点在世界坐标系中的坐标
               A[0],A[1],A[2],               B[0],B[1],B[2]
           };
}}
```

说明

上述工具类中主要就是提供了根据上一小节介绍的思路计算A、B两点在世界坐标系中坐标的calculateABPosition方法。

（3）下面应该开发的是用于拾取计算的AABB包围盒类AABB3，其与前面第16章中介绍AABB包围盒时给出的AABBBox类的大部分代码基本一致。最大的不同就是增加了求物体经仿射变换（平移、旋转、缩放）后AABB包围盒的setToTransformedBox方法，其代码如下。

代码位置：见随书光盘中源代码/第19章/Sample19_1/com/bn/Sample19_1目录下的AABB3.java。

```java
public AABB3 setToTransformedBox(float[] m){      //求仿射变换后物体AABB包围盒的方法
  Vector3f[] va = this.getAllCorners();           //获取原始AABB包围盒所有顶点的坐标
  float[] transformedCorners=new float[24];       //用于存放仿射变换后顶点坐标的数组
  float[] tmpResult=new float[4];                 //临时存储一个点齐次坐标的数组
  int count=0;                                    //计数器
  for(int i=0;i<va.length;i++){                   //对原始包围盒的8个顶点进行遍历
     float[] point=new float[]{va[i].x,va[i].y,va[i].z,1};   //将顶点坐标组织成齐次坐标
     Matrix.multiplyMV(tmpResult, 0, m, 0, point, 0); //对顶点坐标进行仿射变换(乘以变换矩阵m)
     transformedCorners[count++]=tmpResult[0];    //保存变换后顶点的X坐标
         transformedCorners[count++]=tmpResult[1];   //保存变换后顶点的Y坐标
         transformedCorners[count++]=tmpResult[2];   //保存变换后顶点的Z坐标
    }
    return new AABB3(transformedCorners);         //返回变换后的AABB包围盒
  }
```

提示

上述setToTransformedBox方法的思路非常简单：首先得到原始AABB包围盒8个顶点的坐标，然后将这8个顶点的坐标依次通过所属物体的仿射变换矩阵进行变换，最后再求出变换后8个顶点对应的AABB包围盒并返回。这与前面第16章中图16-6介绍的思路完全一致。

（4）接着需要开发的是抽象类TouchableObject，案例场景中所有能够被拾取的物体都需要继承自此类，其代码如下。

代码位置：见随书光盘中源代码/第19章/Sample19_1/com/bn/Sample19_1目录下的TouchableObject.java。

```
1package com.bn.Sample19_1;                         //声明包
2public abstract class TouchableObject {
3    AABB3 preBox;                                  //仿射变换之前的原始 AABB 包围盒
4    float[] m = new float[16];                     //仿射变换(平移、旋转、缩放)矩阵
5    float[] color=new float[]{1,1,1,1};            //顶点颜色
6    float size = 1.5f;;                            //当前物体的放大率
7    public AABB3 getCurrBox(){                     //获得变换后的 AABB 包围盒的方法
8        return preBox.setToTransformedBox(m);      //获取变换后的 AABB 包围盒
9    }
10       public void changeOnTouch(boolean flag){     //根据选中状态改变物体颜色、尺寸的方法
11           if (flag) {                              //若选中
12               color = new float[] { 0, 1, 0, 1 };     //改变颜色为绿色
13               size = 3f;                              //改变尺寸
14           } else {                                    //若没有选中
15               color = new float[] { 1, 1, 1, 1 };     //恢复颜色为白色
16               size = 1.5f;                            //恢复尺寸
17           }
18       }
19       public void copyM(){                            //复制变换矩阵
20        for(int i=0;i<16;i++){
21           m[i]=MatrixState.getMMatrix()[i];
22       }
23   }}
```

- 第3-6行为该类的成员变量，包括仿射变换之前的原始AABB包围盒、仿射变换矩阵、顶点颜色及当前物体的放大率。
- 第10-18行为根据物体的选中状态设置物体的颜色与放大率的changeOnTouch方法。
- 第19-22行为复制物体仿射变换矩阵的copyM方法。之所以要复制物体的仿射变换矩阵是因为MatrixState类中的变换矩阵是随着程序的执行不断变化服务于所有物体的，因此物体应该保留一份仅服务于自身的仿射变换矩阵。

> **提示**
>
> 注意进行矩阵的复制时不能仅复制矩阵的引用，而是应该一一复制矩阵中的元素。

（5）下面需要开发的是表示从obj文件中加载的3D物体对象所属的LoadedObjectVertexNormal Average类，此类与前面第9章中介绍的LoadedObjectVertexNormalAverage类代码基本相同，仅仅改动了3处，具体代码如下。

代码位置：见随书光盘中源代码/第19章/Sample19_1/com/bn/Sample19_1目录下的LoadedObjectVertexNormalAverage.java。

```
1 package com.bn.Sample19_1;                                   //声明包
2 .....//此处省略了引入相关类的一些代码，需要的读者请参考随书光盘
3 public class LoadedObjectVertexNormalAverage extends TouchableObject {
4     .....//此处省略了所有成员变量的声明，需要的读者请参考随书光盘
5     public LoadedObjectVertexNormalAverage(
6         MySurfaceView mv,float[] vertices,float[] normals){
7      initVertexData(vertices,normals);                      //初始化顶点数据
8      initShader(mv);                                        //初始化着色器
9      preBox = new AABB3(vertices);                          //得到原始的AABB包围盒
10       }
11       public void initVertexData(float[] vertices,float[] normals){/*代码省略*/}
12       public void initShader(MySurfaceView mv){/*代码省略*/}
13       public void drawSelf() {
14         copyM();                                           //复制变换矩阵
15         GLES20.glUseProgram(mProgram);                     //指定使用某套着色器程序
16         .....//此处省略了部分代码，需要的读者请参考随书光盘
17       }
18   }
```

提示

3处改动分别为：第3行增加了继承TouchableObject类的代码，第9行增加了得到原始AABB包围盒的代码，第14行增加了复制变换矩阵的代码。

（6）完成了上述工作之后，就可以开发负责整体场景绘制以及触控事件处理的MySurfaceView类了，首先给出该类的代码框架，具体内容如下。

代码位置：见随书光盘中源代码/第19章/Sample19_1/com/bn/Sample19_1目录下的MySurfaceView.java。

```
package com.bn.Sample19_1;                                   //声明包
.....//此处省略了引入相关类的一些代码，需要的读者请参考随书光盘
class MySurfaceView extends GLSurfaceView {
    .....//此处省略了一些成员变量的声明，需要的读者请参考随书光盘
    ArrayList<TouchableObject> lovnList=new ArrayList<TouchableObject>();//可拾取物体列表
    int checkedIndex=-1;       //被选中物体的索引值，即id，没有被选中时索引值为-1
```

```
    public MySurfaceView(Context context) {/*代码省略*/}
   @Override public boolean onTouchEvent(MotionEvent e){ //触控事件的处理方法
        ...../代码省略，将在后面介绍
      }
      public void setCameraPostion() {/*代码省略*/}      //设置摄像机位置的方法
      public void changeObj(int index){                  //改变当前选中物体的方法
          ...../代码省略，将在后面介绍
      }
      private class SceneRenderer implements GLSurfaceView.Renderer {
          LoadedObjectVertexNormalFace pm;              //从 obj 文件中加载的平面
          LoadedObjectVertexNormalFace cft;             //从 obj 文件中加载的长方体
          LoadedObjectVertexNormalAverage qt;           //从 obj 文件中加载的球体
          LoadedObjectVertexNormalAverage yh;           //从 obj 文件中加载的圆环
          LoadedObjectVertexNormalAverage ch;           //从 obj 文件中加载的茶壶
        public void onDrawFrame(GL10 gl) {
            GLES20.glClear(                             //清除深度缓冲与颜色缓冲
                    GLES20.GL_DEPTH_BUFFER_BIT | GLES20.GL_COLOR_BUFFER_BIT);
             MatrixState.setCamera(cx,cy,cz,tx,ty,tz,0,1,0);   //设置摄像机位置
            MatrixState.setLightLocation(100, 100, 100);   //设置光源位置
            pm.drawSelf();                              //绘制平面
            MatrixState.pushMatrix();                   //保护现场
            MatrixState.translate(-30f, 0f, 0);         //平移变换
            MatrixState.scale(cft.size, cft.size, cft.size);   //缩放变换
            cft.drawSelf();                             //绘制长方体
            MatrixState.popMatrix();                    //恢复现场
            ...../此处省略了绘制其他物体的代码，需要的读者请参考随书光盘
        }
        public void onSurfaceChanged(GL10 gl, int width, int height) {/*代码省略*/}
        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
            GLES20.glClearColor(0.3f,0.3f,0.3f,1.0f); //设置屏幕背景色 RGBA
            GLES20.glEnable(GLES20.GL_DEPTH_TEST);//打开深度检测
            GLES20.glDisable(GLES20.GL_CULL_FACE);//关闭背面剪裁
            MatrixState.setInitStack();                 //初始化变换矩阵
            pm=LoadUtil.loadFromFileVertexOnlyFace(     //加载平面
                    "pm.obj", MySurfaceView.this.getResources(),MySurfaceView.this);
            ch=LoadUtil.loadFromFileVertexOnlyAverage( //加载茶壶
                "ch.obj", MySurfaceView.this.getResources(),MySurfaceView.this);
             cft=LoadUtil.loadFromFileVertexOnlyFace(       //加载长方体
                    "cft.obj", MySurfaceView.this.getResources(),MySurfaceView.this);
             qt=LoadUtil.loadFromFileVertexOnlyAverage(    //加载球体
                    "qt.obj", MySurfaceView.this.getResources(),MySurfaceView.this);
             yh=LoadUtil.loadFromFileVertexOnlyAverage(    //加载圆环
                    "yh.obj", MySurfaceView.this.getResources(),MySurfaceView.this);
             //将可拾取物体加入列表
             lovnList.add(ch); lovnList.add(cft); lovnList.add(qt); lovnList.add(yh);
        }
  }}
```

说明

上述MySurfaceView类的代码框架与本书前面很多案例中的并没有很大区别，主要是加载了需要在场景中绘制的物体并且执行了绘制。

（7）最后需要开发的是在上一步骤中省略的onTouchEvent以及changeObj方法，具体代码如下。

代码位置：见随书光盘中源代码/第19章/Sample19_1/com/bn/Sample19_1目录下的MySurfaceView.java。

```
1 @Override
2 public boolean onTouchEvent(MotionEvent e){                         //触控事件处理方法
3    float y = e.getY();                                              //获取触控点的 X 坐标
4    float x = e.getX();                                              //获取触控点的 Y 坐标
5    switch (e.getAction()) {
6    case MotionEvent.ACTION_DOWN:                                    //若是按下动作
7     //计算仿射变换后 A、B 两点在世界坐标系中的坐标
8     float[] AB=IntersectantUtil.calculateABPosition(
9        x, y,                                                        //触控点 X、Y 坐标
10           Sample19_1_Activity.screenWidth,                         //屏幕宽度
11           Sample19_1_Activity.screenHeight,                        //屏幕高度
12           left,top,near,far                                        //近平面参数
13       );
14       Vector3f start = new Vector3f(AB[0], AB[1], AB[2]); //拾取射线线段起点 A 坐标
15       Vector3f end = new Vector3f(AB[3], AB[4], AB[5]);   //拾取射线线段终点 B 坐标
16       Vector3f dir = end.minus(start);                    //拾取射线线段的长度和方向
17       checkedIndex = -1;                  //设置选中物体索引(-1 表示没有选中物体)
18       int tmpIndex=-1;                    //用于记录距离 A 点最近物体的索引值
19       float minTime=1;                    //用于记录列表中物体与拾取射线线段相交的最短时间
20       for(int i=0;i<lovnList.size();i++){              //遍历列表中的物体
21           AABB3 box = lovnList.get(i).getCurrBox();   //获得物体的 AABB 包围盒
22           float t = box.rayIntersect(start, dir, null);        //计算相交时间
23           if (t < minTime) {                       //若时间小于原最短时间
24               minTime = t;                         //更新最短时间
25               tmpIndex = i;                        //更新最近物体索引
26           }}
27           checkedIndex=tmpIndex;                   //记录最近(选中)物体索引
28           changeObj(checkedIndex);                 //改变被选中物体
29       break;
30       case MotionEvent.ACTION_MOVE:                //若是移动动作
31       .....//此处省略根据触控点移动情况改变摄像机位置的代码，需要的读者请参考随书光盘
32       break;
33      }
34      mPreviousY = y;    mPreviousX = x;            //记录触控点 X、Y 坐标
35      return true;
36  }
37  public void changeObj(int index){                 //根据索引改变选中物体
38       if(index != -1){                             //如果有物体被选中
39       for(int i=0;i<lovnList.size();i++){          //遍历所有可拾取物体列表
40           if(i==index){                            //若为选中物体
41             lovnList.get(i).changeOnTouch(true);  //改变物体颜色及放大率到选中状态
42            } else{                                  //若不是选中物体
44              lovnList.get(i).changeOnTouch(false); //改变物体颜色及放大率到未选中状态
45        }}}
46        else{                                        //如果没有物体被选中
47             for(int i=0;i<lovnList.size();i++){     //遍历所有可拾取物体列表
48               lovnList.get(i).changeOnTouch(false); //改变物体颜色及放大率到未选中状态
49   }}}
```

- 第1-36行为用于处理触控事件的onTouchEvent方法，当为按下动作时首先计算出A、B两点在世界坐标系中的坐标，然后生成拾取射线线段AB。接着计算射线线段与每个物体变换后AABB包围盒的相交时

间，并记录最短时间及其对应的物体索引值，最后根据索引值调用changeObj改变被选中物体的颜色及尺寸。

- 第37-49行为根据索引改变选中物体的changeObj方法，其实现非常简单。若有选中物体则将选中物体设置为选中状态，其他物体设置为未选中状态；若没有选中物体则将所有物体设置为未选中状态。

> **提示**
>
> 计算相交时间的具体方法在第15章介绍光线跟踪时已经详细介绍过，这里不再赘述，需要的读者请参考第15章中的相关内容。

19.2　多点触控

风靡全球的iPhone主要得益于两大革命性的创新，一是传感器体感操控的使用，另一个就是多点触控的使用。Android智能平台自然也不例外，与传感器体感操控相关的内容上一章中已经进行了详细介绍，本节将向读者介绍多点触控技术的使用。

19.2.1　案例效果与基本原理

介绍本节案例的具体开发之前，首先需要了解一下本节案例Sample19_2的运行效果与基本原理，首先给出的是案例的运行效果，如图19-5所示。

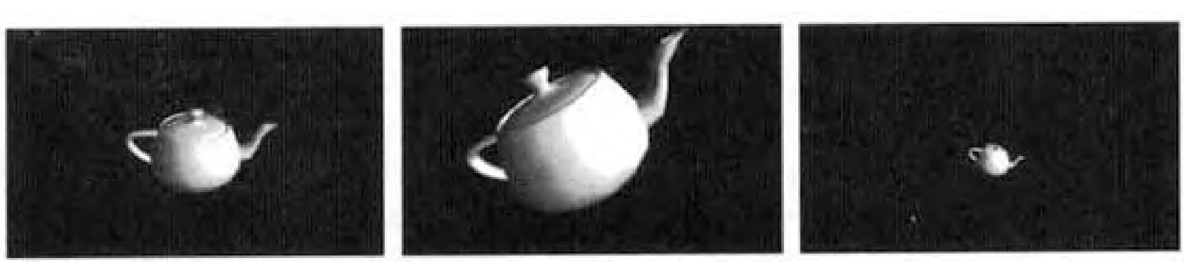


图19-5　案例Sample19_2的运行效果

提示

与本书前面的案例不同，此案例中茶壶的旋转与缩放是采用多点触控技术实现的。案例运行时，用两个手指触摸手机屏幕，当两个手指远离时茶壶被放大，当两个手指接近时茶壶被缩小，当两个手指触点的连线相对于手机屏幕角度变化时茶壶会旋转。要注意的是，仅用一个手指触摸屏幕是不会有任何变化的。

了解了本节案例的运行效果后，下面来介绍多点触控的基本原理。多点触控是指同时支持两个或两个以上的触控点，本节介绍的是同时有两个触控点的情况。这两个触控点一个称之为主点，另一个称之为辅点，每个触控点都有按下、移动、抬起3种动作。

与单点触控相同，触控事件还是在重写的onTouchEvent方法中进行处理，只不过在获取触控动作编号时需要加上掩码与运算，具体代码如下。

```
int action=e.getAction()&MotionEvent.ACTION_MASK;            //获取触控的动作编号
```

动作编号action可以对应多个常量值，分别代表主、辅点的3种动作，如表19-1所列。

表19-1　多点触控中主、辅点的各个动作编号常量

动作编号常量值	含　义
MotionEvent.ACTION_DOWN	主点按下
MotionEvent.ACTION_POINTER_DOWN	辅点按下
MotionEvent.ACTION_MOVE	主、辅点移动
MotionEvent.ACTION_UP	主点抬起
MotionEvent.ACTION_POINTER_UP	辅点抬起

由于多点触控中不止一个触控点，为了管理的方便还需要获取每个触控点的id，具体代码如下。

```
int id=(e.getAction()&MotionEvent.ACTION_POINTER_ID_MASK)//获取触控点id(包括主点以及辅点)
    >>>MotionEvent.ACTION_POINTER_ID_SHIFT;
```

19.2.2　开发步骤

了解了多点触控的基本原理以及案例的运行效果后，本小节将介绍如何进行案例的开发。由于本案例中与绘制相关部分的代码和本书前面很多案例中的基本相同，因此这里仅介绍与多点触控密切相关的部分，具体内容如下。

（1）首先需要开发的是用于封装一个触控点信息的BNPoint类，其代码如下。

代码位置：见随书光盘中源代码/第19章/Sample19_2/com/bn/Sample19_2目录下的BNPoint.java。

```
package com.bn.Sample19_2;                            //声明包
public class BNPoint {
    float oldX; float oldY;                           //上一次此触控点的X、Y坐标
    boolean hasOld = false;                           //是否已经存在上一次触控位置的标志位
    float x; float y;                                 //触控点当前X、Y坐标
    public BNPoint(float x, float y) {                //构造器
        this.x = x; this.y = y;                       //初始化X、Y坐标
    }
    public void setLocation(float x, float y) {       //设置触控点新位置的方法
            oldX = this.x; oldY = this.y;             //把原来位置记录为旧位置
            hasOld = true;                            //设置是否已经存在上一次触控位置的标志位
            this.x = x; this.y = y;                   //设置新位置
        }
        public static float calDistance(BNPoint a, BNPoint b) {//计算两个触控点距离的方法
            float result = 0;
            result = (float) Math.sqrt((a.x - b.x) * (a.x - b.x) +
                                                        //计算两个触控点的直线距离
                    (a.y - b.y) * (a.y - b.y));
            return result;
        }
    }
```

说明

上述BNPoint类非常简单，主要是提供了封装触控点位置坐标以及计算两个触控点直线距离的功能，开发此类是为了后面便于对

多个触控点进行管理。

（2）接着开发的是负责绘制整个3D场景以及执行触控事件处理的MySurfaceView类，其代码如下。

代码位置：见随书光盘中源代码/第19章/Sample19_2/com/bn/Sample19_2目录下的MySurfaceView.java。

```
package com.bn.Sample19_2;                                    //声明包
.....//此处省略了引入相关类的一些代码，需要的读者请参考随书光盘
class MySurfaceView extends GLSurfaceView {
  private SceneRenderer mRenderer;                           //场景渲染器
  .....//此处省略了部分成员变量的声明，需要的读者请参考随书光盘
   HashMap<Integer,BNPoint> hm=new HashMap<Integer,BNPoint>(); //触控点 Map
   float distance=0;                                         //主、辅点距离
   float currScale=2.5f;                                     //初始缩放比例
   float scaleSpeedSpan=100;                                 //缩放步进比例
       float angle=0;                                        //初始旋转角度
       public MySurfaceView(Context context) {/*代码省略*/}
      @Override
      public boolean onTouchEvent(MotionEvent e) {                   //触控事件处理方法
         int action=e.getAction()&MotionEvent.ACTION_MASK;           //获取触控动作编号
         int id=(e.getAction()&MotionEvent.ACTION_POINTER_ID_MASK)   //获取触控点 id
               >>>MotionEvent.ACTION_POINTER_ID_SHIFT;
         switch(action) {
            case MotionEvent.ACTION_DOWN:                     //主点按下
            case MotionEvent.ACTION_POINTER_DOWN:             //辅点按下
              //不论是主点还是辅点按下皆向 Map 中放入一个新点
              hm.put(id, new BNPoint(e.getX(id), e.getY(id)));
              if (hm.size() == 2) {                           //若已经有两个触控点按下
                 BNPoint bpTempA = hm.get(0);                 //从 Map 中取出第一个触控点
                 BNPoint bpTempB = hm.get(1);                 //从 Map 中取出第二个触控点
                 distance = BNPoint.calDistance(bpTempA, bpTempB);//计算两个触控点之间的距离
              }
            break;
            case MotionEvent.ACTION_MOVE:                     //主/辅点移动
              Set<Integer> ks = hm.keySet();                  //获取 Map 的键值集合
              for (int i : ks) {                              //遍历所有触控点
                 hm.get(i).setLocation(e.getX(i), e.getY(i));     //更新触控点位置
              }
```

```
35              if (hm.size() == 2) {                           //若当前有两个触控点按下
36                  BNPoint bpTempA = hm.get(0);                 //从 Map 中取出第一个触控点
37                  BNPoint bpTempB = hm.get(1);                 //从 Map 中取出第二个触控点
39                  float currDis = BNPoint.calDistance(bpTempA, bpTempB); //计算两点距离
40                  currScale = currScale + (currDis - distance) / scaleSpeedSpan;
                                                                 //计算缩放系数
41                  if (currScale > 4 || currScale < 1) {    //若缩放系数超过范围进行恢复
42                      currScale = currScale - (currDis - distance) / scaleSpeedSpan;
43                  }
44                  distance = currDis;                          //记录两点距离
45                   if (bpTempA.hasOld || bpTempB.hasOld) {    //若两个触控点都有上一位置
46                      double alphaOld = Math.atan2((bpTempA.oldY - bpTempB.oldY),
47                          (bpTempA.oldX - bpTempB.oldX));      //计算上一位置对应的角度
48                      double alphaNew = Math.atan2((bpTempA.y - bpTempB.y),
49                          (bpTempA.x - bpTempB.x));        //计算当前位置对应的角度
11                      //两个角度相减得到旋转变化角度，并将旋转变化角度叠加到总旋转角度中
50                      angle = angle - (float) Math.toDegrees(alphaNew - alphaOld);
52                  }}
53              break;
54              case MotionEvent.ACTION_UP:                  //主点抬起
55                  hm.clear();                              //清空触控点 Map
56              break;
57              case MotionEvent.ACTION_POINTER_UP:          //辅点抬起
58                  hm.remove(id);                           //从 Map 中删除对应 id 的辅点
59              break;
60          }
61          return true;
62      }
63      public void setCameraPostion() {/*代码省略*/}     //设置摄像机位置的方法
64      private class SceneRenderer implements GLSurfaceView.Renderer {
65        LoadedObjectVertexNormalAverage ch;            //从 obj 文件中加载的茶壶
66        public void onDrawFrame(GL10 gl){
67            GLES20.glClear(                            //清除深度缓冲与颜色缓冲
68                    GLES20.GL_DEPTH_BUFFER_BIT | GLES20.GL_COLOR_BUFFER_BIT);
69            MatrixState.pushMatrix();                  //保护现场
70            MatrixState.translate(0, -10, 0);          //平移变换
71            MatrixState.scale(currScale, currScale, currScale);//缩放变换
72            MatrixState.rotate(angle, 0, 0, 1);        //旋转变换
73            ch.drawSelf();                             //绘制茶壶
74            MatrixState.popMatrix();                   //恢复现场
75        }
76        public void onSurfaceChanged(GL10 gl, int width, int height) {/*代码省略*/}
77        public void onSurfaceCreated(GL10 gl, EGLConfig config) {/*代码省略*/}
78    }
79  }
```

- 第13-62行为处理触控事件的onTouchEvent方法，对主点和辅点3种动作的处理代码都在此方法中。
- 第18-26行为对主点以及辅点按下动作处理的代码，无论主点还是辅点按下，都创建一个对 应的BNPoint类型的对象放到触控点Map中。

- 第28-53行为对主点以及辅点移动动作处理的代码。其中首先更新每个触控点的位置，然后根据两个触控点的情况计算出新的缩放系数以及旋转角度供绘制部分使用。

> **提示**
>
> 将所有的触控点对象存放到一个Map中是为了便于管理，读者如果有其他更好的思路可以自行对源代码相关部分进行修改。

19.3　多键监听

很多游戏操作的过程中都允许玩家同时按下多个键来进行联合操作，从多年前笔者幼时红白机上的魂斗罗到现在智能手机上的很多游戏都是如此。因此，本小节将向读者介绍如何结合多点触控在Android平台上实现多键监听。

19.3.1　案例效果与基本原理

介绍本节案例的具体开发之前首先需要了解一下本小节案例Sample19_3的运行效果与基本原理，首先给出的是案例的运行效果，如图19-6所示。

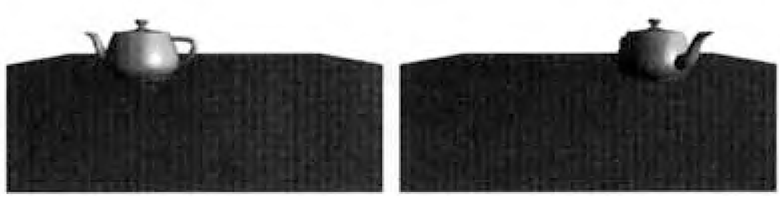


图19-6　案例Sample19_3运行效果

> **提示**
>
> 本案例的场景中有一只茶壶，触摸屏幕上不同的区域可以移动或旋转茶壶。触摸屏幕的左上部分茶壶左移，触摸屏幕的右上部分

茶壶右移，触摸屏幕的左下部分茶壶逆时针旋转，触摸屏幕的右下部分茶壶顺时针旋转，4个区域的分布如图19-7所示。

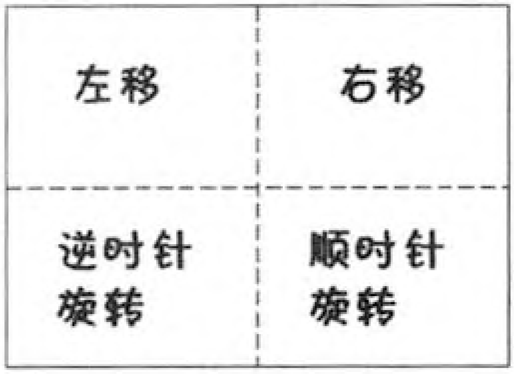


图19-7　屏幕各区域触控功能

虽然Android平台支持多点触控，但并不能直接支持多键监听。也就是说Android平台每个时刻只能处理一个触控事件，故要实现多键监听的效果还需要做一些简单的开发，基本思路如下。

- 首先设置一个int型变量keyState用来记录当前的按键状态（也就是触控状态，在大部分没有键盘的Android智能手机上都是采用触摸屏幕指定区域的方式），当有键（触控点）按下或抬起时在事件处理方法中把对应的状态记录到keyState中对应的比特位。按下某个键时其对应比特位置1，抬起时恢复为0，如图19-8所示。

比特位编号	3	2	1	0
含义	茶壶左移	茶壶右移	逆时针旋转	顺时针旋转
当前值	1	0	1	0

左移的同时逆时针旋转

比特位编号	3	2	1	0
含义	茶壶左移	茶壶右移	逆时针旋转	顺时针旋转
当前值	0	1	0	1

右移的同时顺时针旋转

图19-8　根据按键情况设置对应的比特位

提示

图19-8中只给出了4个键的情况，当有更多的键时可以使用keyState中更多的比特位，一个int型变量有32个比特位，足够使用了。

- 然后需要开启一个后台线程，其定时读取keyState中的状态并执行具体的响应动作。例如，同时读到了茶壶左移以及逆时针旋转比特位为1，则在此次执行动作时既左移又逆时针旋转，这样就非常容易地实现了两个键同时起作用的效果。同时，执行响应动作的频率也可以通过调整线程的休眠时间来调整。

19.3.2　开发步骤

了解了多点监听的基本原理以及案例的运行效果后，本小节将介绍如何进行案例的开发。由于本案例中与绘制相关部分的代码与本书前面很多案例中的基本相同，因此这里仅介绍与多键监听密切相关的部分，具体内容如下。

（1）首先开发的是用于记录一个触控点位置的BNPoint类，其代码如下。

代码位置：见随书光盘中源代码/第19章/Sample19_3/com/bn/Sample19_3目录下的BNPoint.java。

```
package com.bn.Sample19_4;                    //声明包
public class BNPoint{
    float x; float y;                         //触控点 X、Y坐标
    public BNPoint(float x,float y){          //有参构造器
        this.x=x; this.y=y;                   //初始化触控点坐标
} }
```

> **说明**
>
> 本类对象的主要功能是记录触控点的坐标，程序将根据触控点坐标所在区域对相应的比特位进行置0或置1操作。

（2）接着需要开发的是定时读取各比特位状态，并根据各比特位状态执行特定响应动作的线程KeyThread，其代码如下。

代码位置：见随书光盘中源代码/第19章/Sample19_3/com/bn/Sample19_3目录下的KeyThread.java。

```
1 package com.bn.Sample19_4;                         //声明包
2 .....//此处省略了引入相关类的一些代码，需要的读者请参考随书光盘
3 public class KeyThread extends Thread{
4     int keyState=0;                                //存储各比特位的 int 型变量
5     boolean flag=true;                             //线程循环标志
6     @Override
7     public void run(){
8         while(flag){                               //线程不断循环定时读取各比特位值
9             if((keyState&0x1)!=0){                 //茶壶左移动作对应比特
10                    xOffset-=0.3f;
11                }else if((keyState&0x2)!=0){       //茶壶右移动作对应比特
12                    xOffset+=0.3f;
13                }
14                if((keyState&0x4)!=0){             //逆时针旋转动作对应比特
15                    yAngle+=2.5f;
16                }else if((keyState&0x8)!=0){   //顺时针旋转动作对应比特
17                    yAngle-=2.5f;
18                }
19                if(yAngle>=360||yAngle<=-360){ //如果旋转角度绝对值大于 360，则置零
20                    yAngle=0;
21                }
22                try{Thread.sleep(30);}catch(InterruptedException e){e.printStack
                  Trace();}
23        } }
24        public void keyPress(float x,float y){  //检测按下触控点所在区域并置对应比特位值为 1
25            if(x>=0&&x<screenWidth/2&&y>=0&&y<screenHeight/2){
26                keyState=keyState|0x1;          //触控点在茶壶左移区域
27            }else if(x>=screenWidth/2&&x<screenWidth&&y>=0&&y<screenHeight/2){
28                keyState=keyState|0x2;          //触控点在茶壶右移区域
29            }else if(x>=0&&x<screenWidth/2&&y>=screenHeight/2&&y<screenHeight){
30                keyState=keyState|0x4;          //触控点在逆时针旋转区域
31            }else if(x>=screenWidth/2&&x<=screenWidth&&y>=screenHeight/2&&y
              <=screenHeight){
32                keyState=keyState|0x8;          //触控点在顺时针旋转区域
33        } }
34        public void keyUp(float x,float y){     //检测抬起触控点所在区域并置对应比特位值为 0
35            if(x>=0&&x<screenWidth/2&&y>=0&&y<screenHeight/2){
36                keyState=keyState&0xE;          //触控点在茶壶左移区域
37            }else if(x>=screenWidth/2&&x<screenWidth&&y>=0&&y<screenHeight/2){
38                keyState=keyState&0xD;          //触控点在茶壶右移区域
39            }else if(x>=0&&x<screenWidth/2&&y>=screenHeight/2&&y<screenHeight){
40                keyState=keyState&0xB;          //触控点在逆时针旋转区域
41            }else if(x>=screenWidth/2&&x<=screenWidth&&y>=screenHeight/2&&y<=
              screenHeight){
42                keyState=keyState&0x7;          //触控点在顺时针旋转区域
43        } }
44        public void clearKeyState(){//所有触控点均抬起时调用的方法，即将 keyState 值置为 0
45            keyState=keyState&0x0;
46    } }
```

- 第7-23行为线程定时读取各比特位状态并执行特定响应动作的run方法，执行响应动作的频率可以通过修改第22行的线程休眠时间来进行调整。

- 第24-33行为检测按下的触控点所在区域并置对应比特位值为1的keyPress方法。
- 第34-43行为检测抬起的触控点所在区域并置对应比特位值为0的keyUp方法。

> **提示**
>
> 上述keyPress与keyUp方法中是针对本案例中设定的区域进行的判断，实际开发中读者应该根据界面的设计调整这两个方法中区域判断的相关代码。

（3）最后开发的是MySurfaceView类中处理触控事件的onTouchEvent方法，其代码如下。

代码位置：见随书光盘中源代码/第19章/Sample19_3/com/bn/Sample19_3目录下的MySurfaceView.java。

```
@Override
public boolean onTouchEvent(MotionEvent event){                     //触控事件处理方法
    int action=event.getAction()&MotionEvent.ACTION_MASK;           //获取触控动作编号

    int id=(event.getAction()&MotionEvent.ACTION_POINTER_ID_MASK)>>>
        MotionEvent.ACTION_POINTER_ID_SHIFT;          //获取触控点id
    switch(action){
        case MotionEvent.ACTION_DOWN:                 //主点按下
        case MotionEvent.ACTION_POINTER_DOWN:         //辅点按下
            hm.put(id, new BNPoint(event.getX(id),event.getY(id)));
                kt.keyPress(event.getX(id), event.getY(id));//更改对应比特位的值
            break;
            case MotionEvent.ACTION_POINTER_UP:                     //辅点抬起
                float x=hm.get(id).x;                               //获取抬起触控点的x坐标
                float y=hm.get(id).y;                               //获取抬起触控点的y坐标
                hm.remove(id);                                      //移除抬起的触控点
                kt.keyUp(x, y);                                     //更改对应比特位的值
            break;
            case MotionEvent.ACTION_UP:                             //主点抬起
                hm.clear();                                         //清除所有触控点
                kt.clearKeyState();                                 //置所有比特位为0
            break;
        }
        return true;
    }
```

说明

上述onTouchEvent方法与上一节案例中的非常类似，主要是基于多点触控技术调用相应的方法来设置记录当前按键状态的相应各比特位值。另外，采用本案例所介绍的多键监听技术还大大提高了应用程序操控模块的灵活性。例如想将程序修改为由传感器操控只需要在传感器事件处理方法中调用对应的方法设置对应比特位的值即可，完全不会影响程序的其他部分。

19.4　本章小节

本章主要介绍了3D拾取、多点触控以及多键监听等在游戏开发中常用的人机交互技术，通过本章的学习，读者可以在开发时灵活运用这些技术，满足用户的各种需求。

第20章　BN赛艇

伴随着智能手机硬件的升级，手机游戏种类越来越丰富，这使开发人员基于手机开发出更为真实的游戏场景成为了可能。本章将介绍一款基于Android系统的赛艇游戏，该游戏的主要特点为：游戏场景真实、游戏中的碰撞检测准确、游戏场景宏大。

20.1　游戏背景及功能概述

这一节将介绍该游戏的背景，并对该游戏的功能进行基本的介绍。该游戏中的赛艇模仿自F1摩托艇。通过本节的学习，读者将会对该游戏的背景和游戏功能有一定的了解。

20.1.1　背景概述

F1赛艇又称为一级方程式摩托艇，其世界锦标赛是由国际摩托艇联盟（简称国际摩联）于1981年发起组织的。一级方程式摩托艇锦标赛与奥运会、世界杯足球赛以及F1汽车赛一道，被公认为世界上最具影响力以及收视率最高的世界4大体育赛事之一。

F1赛艇与F1赛车相同，也是系列赛事，其惊险性与观赏性在于最高时速为250公里/小时，且从0到100公里/小时加速只需要3.5 s，代表了水上摩托艇运动的最高水平。F1赛艇的驾驶员技术、摩托艇与发动机技术3者结合，代表了人类向速度极限挑战的一种超越，是现代文明中高速度和高科技的象征。

20.1.2　功能介绍

前一小节介绍了本游戏的开发背景，本小节将对该游戏的具体功能进行介绍，具体的步骤如下。

（1）运行本游戏，首先进入的是欢迎界面，分别如图20-1和图20-2所示。



图20-1　欢迎界面1



图20-2　欢迎界面2

（2）当游戏欢迎界面结束以后，将进入本游戏的主菜单界面，在主菜单界面选择不同的选项从而进入不同的界面，如图20-3所示。



图20-3　主菜单界面

（3）在主菜单界面中选择“选船”按钮，将进入选船界面，玩家可以选择自己喜欢的船只，如图20-4所示。选择船时，只需要找到自己喜欢的船，然后按返回键即可选择。

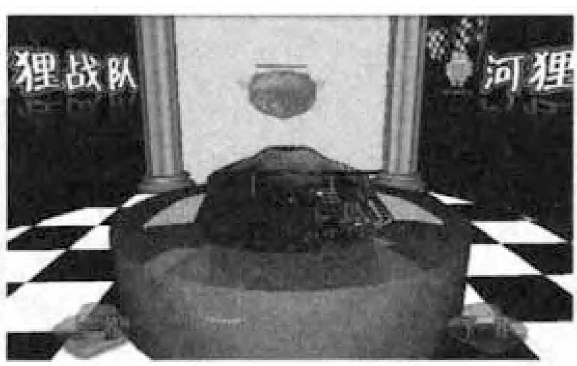


图20-4　选船界面

（4）在主菜单界面中选择“音效设置”按钮，将进入音效设置界面，在该界面中可以打开或者关闭背景音乐和游戏音效，如图20-5所示。

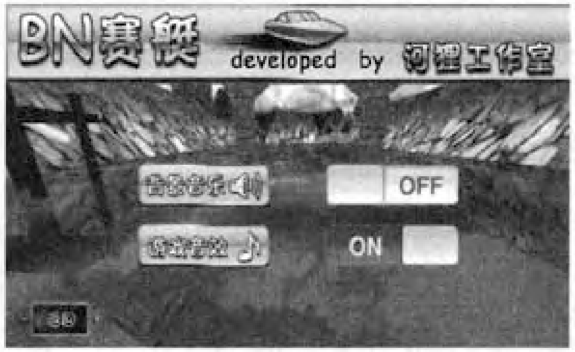


图20-5　音效设置界面

（5）在主菜单界面中选择“游戏帮助”按钮，将进入帮助界面，如图20-6所示。该界面中对本游戏的玩法进行了基本的介绍，单击“下一页”按钮可以进入帮助界面的下一页，单击“上一页”按钮可以进入帮助界面的上一页，单击“返回”按钮返回主菜单界面。



图20-6　帮助界面

（6）在主菜单界面中选择“关于”按钮，将进入关于界面，如图20-7所示。该界面中说明了本游戏的相关版权。

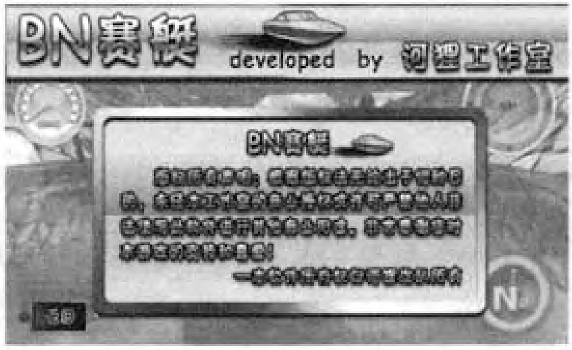


图20-7　关于界面

（7）在主菜单界面中选择“进入游戏”按钮，将进入游戏模式选择界面，如图20-8所示。玩家在该界面中可以选择不同模式的游戏，也可以查看游戏的记录。



图20-8　游戏模式选择界面

（8）在游戏模式界面中选择其中的一种，将进入本游戏的加载界面，如图20-9所示。该界面主要是加载游戏中需要使用到的资源文件。

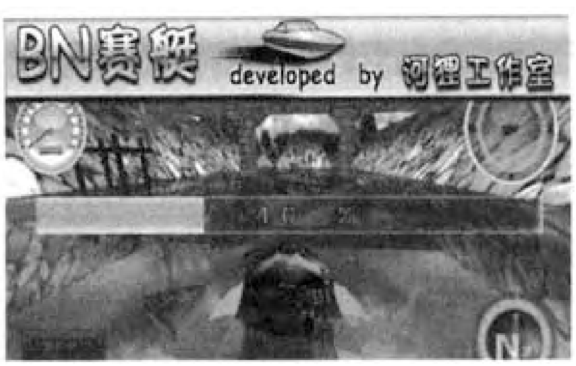


图20-9　加载界面

（9）当加载完本游戏的资源文件后，将进入游戏界面。进入游戏界面后，首先显示的是游戏的倒计时画面，如图20-10所示。



图20-10　游戏开始倒计时

（10）倒计时结束后，将进入本游戏的游戏界面，如图20-11所示。



图20-11　计时模式

（11）在游戏界面中上侧是本游戏的倒视镜，如果玩家感觉添加倒视镜的游戏有碍自己的观察，可以单击该倒视镜区域关闭倒视镜，如图20-12所示。



图20-12　关闭后视镜

（12）如果顺利地跑完本局游戏，本游戏会自动记录游戏用时，同时与以前通过本模式游戏的用时进行比较，判断玩家是否突破纪录，如图20-13所示。



图20-13　计时模式结束

（13）在游戏模式界面中选择“竞速模式”，本游戏会自动进入竞速模式，如图20-14所示。



图20-14　竞速模式

（14）如果玩家在竞速模式中跑完一局游戏，那么游戏也会自动记录游戏用时，同时与其他两艘赛艇的用时作比较，得出自己的游戏排名，如图20-15所示。



图20-15　竞速模式结束

（15）在游戏模式界面中选择“记录查询”，将进入本游戏的记录查看界面，如图20-16所示。在该界面中玩家可以选择不同的模式查看游戏的用时。



图20-16　记录查询

20.2　游戏的策划及准备工作

本小节主要介绍本游戏的策划及开发前的一些准备工作。游戏的开发前需要做一些准备工作，如对将要开发的游戏进行策划、查找合适的图片、音乐以及加载的赛艇等，这样在开发游戏时可以做到事半功倍的效果。

20.2.1　游戏的策划

本小节将对游戏的策划进行简单的介绍，在真实的游戏开发中，该步骤还需更细致、更具体、更全面，该游戏的策划如下所列。

- 游戏类型。

本游戏是一款3D的手机游戏，属于休闲类游戏。

- 运行目标平台。

游戏目标平台为Android 2.2或者更高的版本，同时手机必须有GPU（显卡），因为使用OpenGL ES 2.0的部分工作是在GPU上完成的。

- 目标受众。

本游戏采用移动设备为载体，可以随时随地地打开手机进入游戏。同时使用的3D开发技术使得本游戏场景更加真实逼真，适合儿童、青年人和老年人。

- 操作方式。

本游戏操作简单，使用传感器技术，只需要转动手机即可控制赛艇的运动，这使得本游戏具有很高的可玩性。

- 呈现技术。

本游戏以OpenGL ES 2.0作为游戏呈现技术，同时添加了物体在水中的倒影，这样使得游戏画面更加细腻、真实。

20.2.2　Android平台下游戏的准备工作

本小节将介绍一些开发前的准备工作，包括搜集以及制作图片、声音以及赛艇的3D模型文件（obj文件）等资源，其详细步骤如下。

（1）首先介绍的是本游戏中所用到的图片资源，将所有图片资源放在项目文件下的res/drawable-nodpi文件夹下。由于一次性将所有图片信息全部列出不利于读者观看，所以这里将图片分为2部分列出，第一部分图片的详细情况如表20-1所列。

表20-1　游戏中的部分图片资源

图片名	大小（KB）	像素（w×h）	用　途
about.png	11.3	200×59	未按下状态的关于按钮
about_press.png	9.65	200×59	按下状态的关于按钮
airship_st.png	0.32	128×128	飞艇身体部分图片
airship_wy.png	0.29	128×128	飞艇尾翼部分图片
androidheli.jpg	26.6	480×320	欢迎按钮图片 2
back.png	11.8	125×50	未按下状态的返回按钮
back_press.png	11.4	125×50	按下状态的返回按钮
background.jpg	109	854×480	主菜单界面背景图
bg_bitmap0-4.jpg	105	854×480	帮助界面背景图
bg_music.png	12.4	200×59	未按下的背景音乐按钮
bridge_cm.jpg	64.1	512×256	游戏中桥的贴图
castle0-1.jpg	26.3	256×256	游戏中城堡的贴图
chooseboat.png	13.0	200×59	未按下的选船按钮

续表

图 片 名	大小（KB）	像素（w×h）	用　途
chooseboat_press.png	13.9	200×59	按下状态的选船按钮
close.png	13.8	200×57	关闭声音的按钮图片
d0-9.png	3.58	16×30	记录查询界面数字图片
daojishi.png	35.1	256×64	倒计时界面图片
dialog.9.png	0.94	150×88	对话框背景图
down.png	10.2	128×64	选船界面下一艘按钮图
exit.png	13.9	200×59	未按下的退出游戏图片
exit_press.png	13.2	200×59	按下后的退出游戏图片
floor.png	0.23	128×128	选船界面地板图片
game_end.png	13.5	128×64	游戏结束条幅用图
game_music.png	12.8	200×59	未按下的游戏音效图片
game_start.png	14.3	128×64	游戏开始条幅用图
gamehelp.png	13.2	200×59	未按下游戏帮助图片
gamehelp_press.png	12.8	200×59	按下的游戏帮助图片
gg1-3.jpg	9.05	256×256	选船界面装饰图片
gg_0-3.png	15.0	128×64	游戏中广告用图
gg_sz.png	10.4	64×64	广告牌子下面石柱用图
go.png	85.3	512×256	发射氮气按钮图片
grass.jpg	149	512×512	河道山侧草地用图
guanyu.jpg	122	854×480	关于界面用图
hd_rock.jpg	53.4	256×256	河道山侧山地用图
heli.jpg	29.9	480×320	欢迎界面用图 1
hengxian.png	2.94	16×30	记录查询界面横线用图
icon.png	10.3	72×72	游戏图标
jt_wl.png	0.3	128×128	交通锥用图
load_bj.jpg	47.1	512×256	加载界面背景图片
load_dt.png	29	512×128	加载条背景图
load_fgt.png	4.19	512×128	加载条覆盖图
maohao.png	2.91	16×30	记录查询界面冒号用图
mark0-4.png	0.69	361×43	帮助界面标记图片
mountion_land.png	0.54	16×16	游戏中隧道图
mountion_land_0.png	0.48	16×16	游戏中小山用图

（2）表20-1中介绍了本案例中使用到的部分图片，接下来将介绍剩余的图片，如表20-2所列。

表20-2　游戏中的剩余图片

图片名	大小（KB）	像素（w×h）	用　途
next_page.png	11.3	125×50	未按下的下一页按钮图片
next_page_press.png	10.8	125×50	按下的下一页按钮图片
number.png	11.5	512×32	游戏中数字用图
open.png	14.9	200×57	开启声音按钮
other_boat.png	3.53	64×64	雷达上其他船只标记图
play.png	13.4	200×59	未按下的进入游戏图片
play_press.png	13.0	200×59	按下的进入游戏图片
pre_page.png	11.2	125×50	未按下的上一页按钮图片
pre_page_press.png	10.7	125×50	按下的上一页按钮图片
rador_bg.png	60.5	256×256	游戏中的雷达背景图
rador_plane.png	4.16	32×32	雷达中代表自身的图片
record_select.png	12.8	227×50	未按下的记录查询用图
record_select_press.png	14.1	227×50	按下的记录查询用图
rmode.png	29.8	608×274	竞速模式记录查询背景图
rock.jpg	57.4	512×256	山上石头用图
shache.png	13.7	128×32	刹车按钮用图
shrub-3.png	15.3	128×128	游戏中树木用图
sky.jpg	46.4	1024×128	天空穹用图
soundset.png	13.1	200×59	未按下的音效设置图片
soundset_press.png	12.7	200×59	按下的音效设置图片
speed_down.png	15.3	128×128	减速晶体用图
speed_mode.png	12.7	227×50	未按下竞速模式用图
speed_mode_press.png	15.0	227×50	按下竞速模式用图
speed_up.png	24.1	128×128	加速晶体用图
time.png	64.1	512×256	游戏中的计时用图
timer_mode.png	14.8	227×50	未按下计时模式用图
timer_mode_press.png	15.8	227×50	按下计时模式用图
tmode.png	26.0	608×274	计时模式记录查询背景图
two_ceyi.png	5.03	64×64	第二艘船的侧翼用图
two_chuanshen.png	65.8	256×256	第二艘船船身用图
two_pentong.png	6.20	64×64	第二艘船的喷筒用图

续表

图片名	大小（KB）	像素（w×h）	用　途
two_shangmian.png	18.5	128×128	第二艘船的上侧用图
two_weijia.png	0.17	64×64	第二艘船的尾翼下侧用图
two_weiyi.png	6.71	64×64	第二艘船的尾翼用图
up.png	9.94	128×64	选船界面上一艘按钮用图
water.png	11.7	64×64	水面用图
wood.png	22.5	128×128	船埠头用图
xd.png	0.14	16×16	河道中的小山的灰度图
ybp.png	19.1	128×128	仪表盘用图

（3）本游戏中还将部分图片资源存储在res/raw目录下，如表20-3所列。

表20-3　res/raw目录下的图片资源

图片名	大小（KB）	像素（w×h）	用　途
houshijing.png	11.1	256×64	后视镜用图
one_chuanshen.png	63.2	512×512	第一艘船的船身用图
one_dizuo.png	18.3	256×256	第一艘船的底座用图
one_houmian.png	66.9	256×256	第一艘船的后侧用图
one_qiandizuo.png	13.5	128×128	第一艘船的前侧底座用图
one_qianmian.png	11.3	128×128	第一艘船的前侧用图
three_chuanshen.png	341	512×512	第三艘船的船身用图
three_liangce.png	60.0	256×256	第三艘船的两侧用图
three_pentong.png	12.8	128×128	第三艘船的喷筒用图
three_shangmian.png	269	1024×1024	第三艘船的上侧用图
three_shangmianqian.png	95.2	256×256	第三艘船的上侧前方用图
three_weiyi.png	45.6	256×256	第三艘船的尾翼用图
weilang.png	17.1	128×128	尾浪用图

（4）本游戏中添加了声音，这样游戏更加真实。其中需要将声音资源放在项目目录中的res/raw文件夹下，其详细情况如表20-4所列。

表20-4　声音资源列表

声音文件名	大小（KB）	格　式	用　途
backsound.mp3	157	mp3	游戏背景音乐
boatgo.mp3	19.4	mp3	船开启的声音
daojishi.mp3	4.53	mp3	倒计时音乐
eatthings1.mp3	3.21	mp3	吃东西音乐
pengzhuang.mp3	3.03	mp3	碰撞河道音乐
start.mp3	6.31	mp3	加速音乐
zhuangfei.mp3	2.65	mp3	撞飞物体音乐

（5）本游戏用到的赛艇都是通过3DMax生成obj文件，然后加载的。而生成的赛艇obj文件需要放在项目目录中的assets文件下。其详细情况如表20-5所列。

表20-5　模型文件清单

文件名	大小（KB）	格　式	用　途
one_chuanshen.obj	58.9	obj	第一艘船船身模型
one_dizuo.obj	3.28	obj	第一艘船底座模型
one_houmian.obj	20.5	obj	第一艘船后侧模型
one_qiandizuo.obj	39.8	obj	第一艘船前侧底座
one_qianmian.obj	9.58	obj	第一艘船前侧模型
rome.obj	13.7	obj	选船界面石柱模型
three_chuanshen.obj	28.9	obj	第三艘船船身模型
three_liangce.obj	7.39	obj	第三艘船两侧模型
three_pentong.obj	11.7	obj	第三艘船喷筒模型
three_shangmian.obj	1.15	obj	第三艘船上侧模型
three_shangmianqian.obj	2.50	obj	第三艘船上侧前方模型
three_weiyi.obj	7.36	obj	第三艘船尾翼模型
two_ceyi.obj	22.8	obj	第二艘船两侧模型
two_chuanshen.obj	20.1	obj	第二艘船船身模型
two_pentong.obj	16.3	obj	第二艘船喷筒模型
two_shangmian.obj	15.9	obj	第二艘船上侧模型
two_weijia.obj	8.57	obj	第二艘船尾翼下侧支柱模型
two_weiyi.obj	4.64	obj	第二艘船尾翼模型

20.3　游戏的架构

本小节主要介绍本游戏的类框架，通过本节的介绍，读者应该可以进一步了解本游戏的开发过程，为后面将要开发的代码部分打下坚实的基础。

20.3.1　各个类简要介绍

为了使读者可以更好地了解各个类的作用，本小节将本游戏中用到的所有类分为4部分进行介绍，而每个类的代码将在后面的小节中相继给出。

1．公共类

MyActivity为本游戏中的主控制类，该类继承自Activity，其是整个游戏的控制器，同时也是整个案例的入口。

2．2　D界面相关类

- 欢迎界面类WelcomeView。

本类是欢迎界面类，该类主要是在游戏开始时调用。

- 2D界面的父类MySFView。

本类是一些2D界面类的父类，如主菜单界面MenuView、游戏模式界面GameModeView、关于界面GuanYuView、帮助界面HelpSurfaceView、声音设置界面SoundSurfaceView以及游戏记录界面DSurfaceView均继承自该类。

- 主菜单界面MenuView。

本类为主菜单界面，在该界面中可以通过单击按钮进入其他界面，如选船界面、帮助界面、关于界面、声音选择界面以及游戏模式选择界面等。

- 游戏模式界面GameModeView。

本类为游戏模式界面，在该界面中玩家可以选择不同模式的游戏，也可以选择查看记录来查看本游戏已经记录的数据信息。

- 关于界面GuanYuView。

本界面为关于界面，在该界面中玩家可以查看本游戏的版权说明。

- 帮助界面HelpSurfaceView。

本类为帮助界面，在该界面中玩家可以查看本游戏的使用方法以及基本玩法。

- 声音设置界面SoundSurfaceView。

本类为声音设置界面，在该界面中玩家可以设置本游戏的背景音乐以及游戏的碰撞与吃东西音效。

- 绘制界面类ViewForDraw。

本类为绘制界面类，主要是绘制MySFView类的子类中的一种。

- 数据库类DBUtil。

本类为数据库类，其主要作用是创建数据库，添加数据以及读取数据信息。

- 游戏纪录界面DsurfaceView。

本类为游戏纪录界面类，该类的主要作用是查看“计时模式”的用时或者“竞速模式”用时。

- 是否打破纪录对话框BreakRecordOrNotDialog。

本类为是否打破纪录对话框类，该对话框主要是在一局“计时模式”结束后调用，显示玩家是否成功打破纪录。

- 名次对话框RankingDialog。

本类为名次对话框，该对话框主要是在一局“竞速模式”结束后调用，显示玩家在本局中所处的名次信息。

- 是否有GPU的对话框CheckVersionDialog。

本类为是否有GPU的对话框，如果机器没有GPU，则会弹出本对话框。这样做的主要原因是OpenGL ES 2.0编写的程序不能运行在没有GPU的手机上。

- Android系统版本是否满足要求对话框AndroidVersionDialog。

本类为Android系统版本是否满足要求的对话框，如果Android系统版本低于2.2，那么会弹出本对话框。这是因为OpenGL ES 2.0的程序最低要求是Android 2.2版本。

3．选船界面相关类

- 常量类Constant。

本类是常量类，主要存储部分常量，加载图片以及缩放图片。

- 着色器管理类ShaderManager。

本类主要是将游戏中用到的着色器程序管理起来，使得整个游戏开发中对这方面资源的管理大大简化，同时也减轻了工作量。

- 围墙类ColorLightRect。

本类是围墙类，在选船界面中的围墙有部分是没有贴纹理图，只是通过光照实现的白墙效果。

- 纹理矩形TextureRect。

本类是纹理矩形类，主要作用是用于绘制按钮图片、地板和纹理墙。

- 绘制房间的类HouseForDraw。

本类为绘制房间的类，主要是将颜色围墙、纹理围墙以及石柱这些物体整体组成一个房间。

- 展台的圆面类CircleForDraw。

本类为展台的圆面类，主要是在绘制展台时使用。

- 圆柱类CylinderTextureByVertex。

本类为圆柱类，主要是在绘制展台下侧圆柱时使用。

- 展台类DisplayStation。

本类为展台类，主要是通过绘制展台下侧的圆柱以及圆面从而形成展台。

- 法向量类Normal。

本类为法向量类，该工具类的主要作用是将法向量求平均值。

- 加载物体的类LoadUtilXC。

本类为加载物体类，该类的作用主要是加载指定的obj文件，本案例中该类主要是加载四周的柱子。

- 梅花柱类LoadedObjectVertexNormalXC。

本类为赛艇选择界面中的梅花柱类，该类的主要作用是绘制加载的物体。

- 加载物体类LoadUtilTexXC。

本类为加载物体类，该类的作用主要是加载赛艇的obj文件供选船界面使用。

- 赛艇部件类LoadedObjectVertexTexXC。

本类为赛艇选择界面中赛艇部件类，该类的主要作用是绘制加载的赛艇。

- 赛艇类Boat。

本类为赛艇类，主要功能是专为将赛艇的每个部件进行绘制，这样就获得一艘完整的赛艇。

- 选船界面XCSurfaceView

本类为选船界面类，该类主要是进行赛艇的绘制、上一搜按钮以及下一艘按钮的绘制。

4．游戏界面相关类

- 存储系统矩阵状态的类MatrixState。

该类为存储系统矩阵状态的类，其用于存储当前的变换矩阵、摄像机矩阵、总变换矩阵等，主要实现了3D世界中与变换相关的数学模型。

> **说明**
>
> 由于在本书第4章中对MatrixState类已进行过介绍，所以本章不再对其进行介绍，读者可以到第4章查看相关知识或者查看随书光盘中的源代码。

- 传感器类RotateUtil。

本类为传感器类，该类中主要是通过矩阵更改手机的方向。

- 着色器加载类ShaderUtil。

该类加载对应的Shader程序，包括顶点着色器与片元着色器，然后生成对应的程序，供显卡使用。

> **说明**
>
> 由于在本书中的第3章对该类已经介绍过了，所以本章就不在对其进行介绍，读者可以到第3章查看相关知识或者查看随书光盘中的源代码。

- 常量类Constant。

本类为常量类，主要是存储一些常量、赛艇自动开的数组以及地图矩阵。

- 着色器管理类ShaderManager。

本类主要是将游戏中用到的着色器程序管理起来，使得整个游戏开发中对这方面资源的管理大大简化，同时也减轻了工作量。

- 3D部件数据类TDObjectData。

本类为3D部件数据类，该类主要是存储游戏中所用到的3D部件的位置信息。

- 可撞物体数据类KZBJData。

本类为可撞物体数据类，该类主要是存储游戏中所用到的可碰撞物体的位置信息。

- 可吃物体数据类KeatData。

本类为可吃物体数据类，该类主要是存储游戏中所用到的可吃物体的位置信息。

- 树木数据类TreeData。

本类为树木数据类，该类主要是存储游戏中所用到的树木的位置信息。

- 进度条大背景类BackGround。

本类为进度条界面中的背景类，同时也可以用于数字以及百分号的绘制。

- 进度条类Process。

本类为进度条类，用于进度条背景图以及进度条覆盖图的绘制。

- 倒计时类DaoJiShiForDraw。

本类为倒计时类，该类主要是在程序开始时调用。

- 3D物体父类BNDrawer。

本类为3D物体的父类，这样设计有利于物体的开发，便于管理。

- 飞艇类AirShip。

本类为飞艇类，其继承自BNDrawer。

- 半崖子类B_YZ。

本类为半崖子类，其继承自BNDrawer。

- 桥类Bridge。

本类为桥类，其继承自BNDrawer。

- 城堡类Castle。

本类为城堡类，其继承自BNDrawer。

- 船埠头类Dock。

本类为船埠头类，其继承自BNDrawer。

- 小山类Mountion。

本类为小山类，其继承自BNDrawer。

- 广告类Poster。

本类为广告类，其继承自BNDrawer。

- 隧道类Tunnel。

本类为隧道类，其继承自BNDrawer。

- 3D物体控制类TDObjectForControl。

本类为3D物体控制类，其主要作用是绘制3D物体位置列表中的各个3D物体。

- 船部件模型管理类BoatInfo。

本类为船部件模型管理类，该类主要是将每个赛艇的obj文件单独管理。

- 法向量类Normal。

本类为法向量类，该工具类的主要作用是将法向量求平均值。

- 加载物体类LoadUtil。

本类为加载物体类，该类的作用主要是加载指定的obj文件，本案例中该类主要是加载赛艇。

- 加载后的物体类LoadedObjectVertexNormal。

本类为加载后的物体类，主要是绘制游戏界面中的赛艇。

- 游戏中的赛艇类Boat。

本类为游戏中的赛艇类，主要是对游戏中加载的赛艇进行管理，并绘制。

- 可碰撞物体父类KZBJDrawer。

本类为可碰撞物体父类，与3D物体的父类BNDrawer相似，使用此设计有利于可碰撞物体的 开发，便于管理。

- 交通锥类Cone。

本类为交通锥类，其继承自KZBJDrawer。

- 交通柱类TrafficCylinder。

本类为交通柱类，其继承自KZBJDrawer。

- 可碰撞物体的控制类KZBJForControl。

本类为可碰撞物体的控制类，其主要作用是绘制可碰撞物体位置列表中的各个可碰撞的物体。

- 检查碰撞物体的线程ThreadColl。

本类为每隔一段时间检查碰撞物体的线程，主要是检测可碰撞物体是否与赛艇发生碰撞。

- 可吃物体父类SpeedForEat。

本类为可吃物体父类，与可碰撞物体父类KZBJDrawer以及3D物体的父类BNDrawer相似，使用此设计有利于可碰撞物体的开发，便于管理。

- 可吃晶体类Speed。

本类为可吃晶体类，其继承自SpeedForEat。

- 可吃晶体控制类SpeedForControl。

本类为可吃晶体的控制类，其主要作用是绘制可吃晶体位置列表中可吃的晶体。

- 晶体旋转线程RotateThread。

本类为晶体的旋转线程，通过该线程使得晶体以固定的速度转动。

- 可吃晶体检测碰撞的线程ThreadForEat。

本类为可吃晶体检测碰撞的线程，其主要作用是检测可吃晶体是否与赛艇发生碰撞。

- 灌木纹理矩形类ShrubForDraw。

本类为灌木纹理矩形类，该类的主要作用是绘制单面的灌木。

- 单个灌木比较类MyComparable。

本类为单个灌木比较类其主要作用是比较每片灌木，找出与摄像机距离较近的单片灌木。

- 单个灌木类SingleShrub。

本类为单个灌木类，该类主要是绘制多片灌木，从而形成较为真实的树木。

- 灌木控制类ShrubForControl。

本类为灌木控制类，该类主要作用是比较每个树木距离摄像机的距离，获取距离摄像机的最近树木。

- 河道类RaceTrack。

本类为河道类，主要是通过计算绘制游戏中的河道。

- 检测赛艇与桥碰撞的类PZZ。

本类为检测赛艇与桥碰撞的类，由于桥分为竖直以及横向的，所以在检测时需要同时检测赛艇与竖直桥碰撞以及赛艇与横向桥的碰撞。

- 路径类PathUtil。

本类为路径类，在船自动开时调用，主要是根据赛艇所处河道的编号选择相应的路径。

- 尾浪类WeiLang。

本类为尾浪类，该类主要是在赛艇后侧绘制尾浪，从而使得游戏更加真实。

- 水面类Water。

本类为水面类，本案例中并未使用水面的波动，而使用的是水面流动的模拟。

- 开始与结束条幅类StartAndEnd。

本类为开始与结束条幅类，主要是在开始时绘制开始条幅以及在结束时绘制结束条幅。

- 天空穹类Sky。

本类为天空穹类，添加天空穹效果可以更加真实地模拟现实世界。

- 绘制加速按钮类Dashboard。

本类为绘制加速按钮类，该类的作用是绘制加速按钮，该按钮主要是在有氮气的情况下单击该按钮使得赛艇加速。

- 绘制时间、计圈、氮气、暂停、换人称视角以及加速按钮的类DrawTime。

本类为绘制时间、计圈、氮气、暂停、换人称视角以及加速按钮的类。

- 切割类ClipGrid。

本类为切割类，由于手机自身的限制，所以不可能将地图中的所有全部加载，所以通过本类加载时只能加载其中的部分。

- 模拟键盘监听线程KeyThread。

本类为模拟的键盘监听线程，该类的主要作用是判断赛艇是否与赛道碰撞、判断是否跑完一圈以及判断是否破纪录等。

- 游戏界面类MyGLSurfaceView。

本类为游戏界面类，主要是通过内部类渲染3D场景，绘制游戏界面。

20.3.2　游戏框架简介

前一小节已经介绍了本游戏中用到的所有类，但读者可能对每个类的理解并不深刻，因此本小节将从游戏的整体架构进行介绍，如图20-17所示。

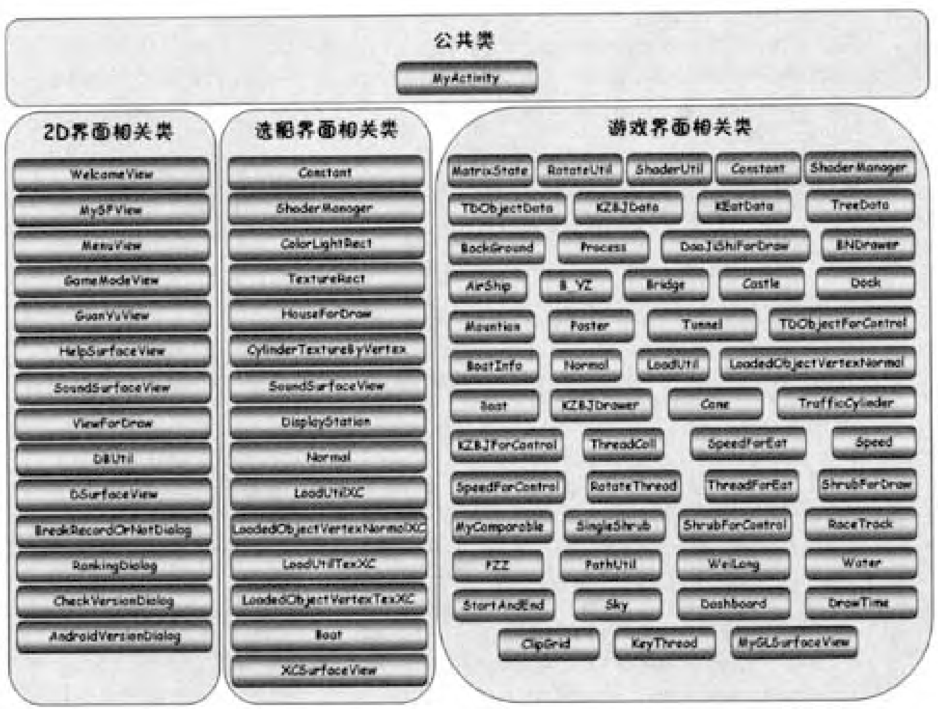


图20-17　游戏整体架构

从图20-17中可以看出本游戏中使用到的类非常多，因此不可能将所有的类依次介绍，所以接下来将按照程序运行的顺序介绍部分类的作用以及游戏的整体框架，具体步骤如下。

（1）打开本游戏，首先进入本游戏的主控制类MyActivity，在该类中首先跳转到主菜单界面MenuView。

（2）在主菜单界面MenuView中玩家可以单击“选船”按钮进入选船界面XCSurfaceView，从而选择自己喜欢的船只；可以单击“音效设置”按钮进入音效界面SoundSurfaceView设置本游戏的音效；可以单击“游戏帮助”按钮进入游戏帮助界面HelpSurfaceView查看游戏的玩法。

（3）在主菜单界面MenuView中还可以单击“关于”按钮进入关于界面GuanYuView查看本游戏的基本版权，同时也可以单击“进入游戏”按钮进入游戏模式选择界面GameModeView选择要进入的游戏模式。

（4）进入游戏模式界面之后，玩家可以选择“计时模式”或者“竞速模式”进入游戏界面MyGLSurfaceView，或者选择“记录查询”进入DSurfaceView。

（5）在主菜单界面MenuView中用户可以单击“退出”游戏按钮退出本游戏，或者直接选择返 回键退出游戏。

说明

开发本案例时，笔者将本案例中的类按照目的不同分为多个包进行管理，这样有助于程序的开发，避免一个包中有过多的类导致维护困难。读者在开发实际项目的过程中也应该根据自己的需要将项目中的类分为多个的包进行管理。

20.4　主控制类MyActivity

从本小节开始将介绍本案例代码的开发，首先介绍本案例的主控制类，该类在程序开始时执行，同时也控制所有页面之间的跳转，主要开发步骤如下。

（1）首先开发主控制类MyActivity的框架，理解该框架有助于读者对于本类的整体了解，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/d2目录下的MyActivity.java。

```
1 package com.bn.st.d2;                                    //声明包名
2 ……//此处省略了本类中导入类的代码，读者可以自行查阅随书光盘中的源代码
3 enum WhichView{                                          //界面的枚举类
4     WELCOME_VIEW,MENU_VIEW,XC_VIEW,GAME_VIEW,HELP_VIEW,
5     SOUND_VIEW,GAME_MODE_VIEW,GUANYU_VIEW,RECORD_VIEW
6 };
7 public class MyActivity extends Activity{          //创建继承Activity的主控制类
8     static WhichView curr;                         //枚举类的引用
9     ……//此处省略了本类中成员变量的声明，读者可以自行查阅随书光盘中的源代码
10        SharedPreferences sp;         // SharedPreferences的引用，用于简单数据的存储
11        ……//此处省略了开发传感器监听器的代码，读者可以自行查阅随书光盘中的代码
12        @Override
13        protected void onCreate(Bundle savedInstanceState){
14            super.onCreate(savedInstanceState);
15            sp=this.getSharedPreferences("actm", Context.MODE_PRIVATE);
16            com.bn.clp.Constant.BgSoundFlag = sp.getBoolean("bgSoundFlag", true);
17            com.bn.clp.Constant.SoundEffectFlag = sp.getBoolean("soundEffectFlag", true);
18            chushihuaSounds();                              //初始化声音资源
19            requestWindowFeature(Window.FEATURE_NO_TITLE);//设置为全屏
20            getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN ,
21                    WindowManager.LayoutParams.FLAG_FULLSCREEN);
22            setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
23            setVolumeControlStream(AudioManager.STREAM_MUSIC);//游戏过程中允许调整音量
24            DBUtil.createTable();                          //创建数据库表
25            ……//此处省略了本方法中的部分代码，读者可以自行查阅随书光盘中的源代码
26        }
27        ……//此处省略了跳转到游戏中部分界面的方法，读者可以自行查阅随书光盘中的源代码
28        ……//此处省略了跳转到选艇界面的方法，将在下面进行介绍
29        ……//此处省略了跳转到计时模式与竞速模式的代码，将在下面进行介绍
30        ……//此处省略了跳转到记录查询界面的代码，读者可以自行查阅随书光盘中的源代码
31        ……//此处省略了游戏重新开始时初始化静态参数的方法，读者可以自行查阅随书光盘中的源代码
32        ……//此处省略了在游戏中退出调用的方法，读者可以自行查阅随书光盘中的源代码
33        ……//此处省略了重写的onResume与onPause方法，将在下面进行介绍
34        ……//此处省略了冲洗的onKeyDown方法，读者可以自行查阅随书光盘中的源代码
35        ……//此处省略了创建声音以及播放声音的方法，读者可以自行查阅随书光盘中的源代码
36        ……//此处省略了获取OpenGLES所支持的最高版本，将在下面进行介绍
37        ……//此处省略了创建对话框与更新对话框的方法，将在下面进行介绍
38    }
```

- 第3-6行为枚举内部类，该类的主要作用是标识当前所处的页面，这样在跳转时就可以根据当前所处的页面选择相应的跳转页面。
- 第15-17行为获取SharedPreferences对象，并根据该对象获取背景音乐以及其他碰撞音乐的boolean值，即判断是否打开背景音乐或者其他碰撞音乐。
- 第19-22行为设置在本游戏中屏幕的模式，屏幕模式为全屏，并且为横屏模式。

（2）由于本类中的跳转界面比较多，不可能对所有的跳转进行一一介绍，所以这里只介绍部分具有代表性的界面跳转代码的开发，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/d2目录下的MyActivity.java。

```
public void gotoXCView(){                              //跳转到选艇界面
    vfd.flag=false;                                    //2D 界面刷帧线程标志位为 false
    xcV = new XCSurfaceView(MyActivity.this);          //创建选船界面对象
    xcV.index_boat=BoatInfo.cuttBoatIndex;             //获取当前船只的索引值
    setContentView(xcV);                               //跳转界面
    xcV.requestFocus();                                //获取焦点
    xcV.setFocusableInTouchMode(true);                 //设置为可触控
    curr=WhichView.XC_VIEW;                            //设置当前界面的索引
}
```

> **说明**
>
> 在跳转到选艇界面之前，首先需要将2D界面的刷帧线程设置为false，然后创建选船界面的对象，并调用setContentView方法跳转到选船界面，最后设置当前界面的索引值。

（3）接下来开发跳转到计时模式游戏界面的方法，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/d2目录下的MyActivity.java。

```
public void gotoTimerGameView(){                       //跳转到计时模式游戏界面
    vfd.flag=false;
    restartGame();                                     //重置游戏中参数的方法
    com.bn.clp.Constant.isSpeedMode=false;             //设置游戏模式选择的相关标志位
    KeyThread.otherBoatFlag=false;                     //是否为多船的标志位
    if(BgSoundFlag&&beijingyinyue==null){              //判断是否需要播放音乐
        beijingyinyue=MediaPlayer.create(this,com.bn.R.raw.backsound);
        beijingyinyue.setLooping(true);                //音乐是否循环的标志位
        beijingyinyue.start();                         //播放音乐
        }
        gameV = new MyGLSurfaceView(MyActivity.this);
        gameV.requestFocus();                          //获取焦点
        gameV.setFocusableInTouchMode(true);           //设置为可触控
        setContentView(gameV);
        curr=WhichView.GAME_VIEW;                      //设置当前界面的索引值
    }
```

说明

由跳转到计时模式游戏界面与跳转到选艇界面方法比较可知，跳转界面都是调用的setContentView方法，但是由于界面的不同，因此需要设置每个界面的相关属性值。如：跳转到计时模式游戏界面方法中就应该将选船标志位设置为false，同时判断是否需要播放音乐。

（4）接下来开发onResume与onPause方法，这两个方法均为继承Activity后重写的方法，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/d2目录下的MyActivity.java。

```
1 @Override
2 public void onResume(){                              //继承Activity重写的onResume方法
3    super.onResume();
4    mySensorManager.registerListener(                 //注册监听器
5          mySensorListener,                           //监听器对象
6          mySensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION),//传感器类型
7          SensorManager.SENSOR_DELAY_GAME
8    );
9    Constant.threadFlag=true;                         //水面换帧线程工作标志位设置为true
10       if(gameV!=null){                              //判断MyGLSurfaceView对象是否为空
11            gameV.kt.moveFlag=true;
12   } }
13   @Override
14   public void onPause(){                            //继承Activity重写的onPause方法
15       super.onPause();
16       mySensorManager.unregisterListener(mySensorListener);    //取消注册监听器
17       Constant.threadFlag=false;                    //水面换帧线程工作标志位设置为false
18       if(gameV!=null){                              //判断MyGLSurfaceView对象是否为空
19            gameV.kt.moveFlag=false;
20   } }
```

- 第2-12行为继承Activity后重写的onResume方法，其主要是在程序由暂停态进入运行态时调用。因此在该方法中需要开启传感器监听，并设置模拟键盘线程的标志位为true。
- 第14-20行为继承Activity重写的onPause方法，该方法主要是在程序由运行态跳转到暂停态时调用，取消传感器监听器，并设置模拟键盘线程的标志位为false。

（5）由于支持OpenGL ES 2.0的Android系统版本必须在Android 2.2（包括2.2）以上，因此开发使用OpenGL ES 2.0的应用程序时需要判断当前系统的版本号是否满足需求，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/d2目录下的MyActivity.java。

```
public int getGLVersion(){                      //获取 OpenGLES 所支持的最高版本
    ActivityManager am = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
    ConfigurationInfo info = am.getDeviceConfigurationInfo();
    int majorVersion=info.reqGlEsVersion;       //获取系统支持的最高版本号(16 进制)
    majorVersion=majorVersion>>>16;             //获取系统支持的最高版本号(10 进制)
    return majorVersion;                        //返回系统支持的最高版本号
}
```

说明

通过ConfigurationInfo对象调用reqGlEsVersion方法获取的是16进制的系统支持的最高版本号，因此需要将其转化为10进制的，这样便于识别。

（6）本案例中在判断系统版本号、判断是否具有GPU、显示计时模式结果以及竞速模式结果时使用的都是对话框，因此接下来将介绍主控制类中与对话框相关方法的开发，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/d2目录下的MyActivity.java。

```
public Dialog onCreateDialog(int id){    //创建对话框时调用的方法
    Dialog result=null;                  //Dialog 的引用
    switch(id){                          //根据对话框的 id 创建不同的对话框
        case 1:
            cvDialog=new CheckVersionDialog(this);
            cvDialog.setCancelable(false); //设置对话框是否可以退出
            result=cvDialog;             //返回版本号对话框
        break;
        ……//此处省略了创建其他对话框的代码，读者可以自行查阅随书光盘中的源代码
```

```
10          }
11          return result;
12      }
13      public void onPrepareDialog(int id, Dialog dialog){
                                                        //弹出对话框调用以动态更新对话框内容的方法
14          switch(id){                                 //若不是等待对话框则返回
15              case 1:
16                Button bok=(Button)cvDialog.findViewById(R.id.ok_button);
17                bok.setOnClickListener(               //为对话框中的按钮添加监听器
18                     new OnClickListener(){
19                         @Override
20                         public void onClick(View v){//重写的onClick方法
21                         Settings.System.putInt(MyActivity.this.getContentResolver(),
22                             Settings.System.ACCELEROMETER_ROTATION,flag);
23                             System.exit(0);   //退出虚拟机
24                   } } );
25              break;
26              ……//此处省略了其他情况更新对话框内容的代码，读者可以自行查阅随书光盘中的源代码
27      } }
```

- 第1-12行为创建对话框时调用的方法，首先需要根据对话框的id判断应该创建的对话框，设置是否可以通过返回键退出，最后返回Dialog对象。
- 第13-27行为弹出对话框时被调用以动态更新对话框内容的方法，该方法也是需要根据对话框id判断需要更新内容的对话框，弹出对话框，并设置对话框中的内容，并对对话框中的按钮添加监听。

20.5　2D界面相关类

本小节将介绍游戏中至关重要的2D界面相关类，该2D相关界面中主要包括部分2D界面以及数据库工具类。该2D界面主要是在游戏未开始之前对游戏的一些设置以及游戏相关内容的查看。一个美观大方的2D界面也是在游戏中比较吸引玩家的。

20.5.1　欢迎界面类WelcomeView

本类是欢迎界面类，本游戏中的欢迎界面采用逐渐消失的方式，游戏开始运行时首先显示的就是该界面。具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/d2目录下的WelcomeView.java。

```
package com.bn.st.d2;                              //声明包名
……//此处省略了本类中导入类的代码，读者可以自行查阅随书光盘中的源代码
public class WelcomeView extends SurfaceView implements SurfaceHolder.Callback{
    MyActivity activity;                           //主控制类的引用
    ……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
    float currentY;                                //当前图片起始点的 y 坐标
    public WelcomeView(MyActivity activity){
        super(activity);
```

```
9          this.activity = activity;                          //创建主控制类的对象
10             this.getHolder().addCallback(this);            //设置生命周期回调接口的实现者
11             paint = new Paint();                           //创建画笔
12             paint.setAntiAlias(true);                      //打开抗锯齿
13             logos[0]=BitmapFactory.decodeResource(activity.getResources(), R.
               drawable.heli);
14             logos[1]=BitmapFactory.decodeResource(activity.getResources(), R.
               drawable.androidheli);
15         }
16         public void onDraw(Canvas canvas){                 //绘制方法
17             paint.setColor(Color.BLACK);                   //设置画笔颜色
18             paint.setAlpha(255);                           //设置不透明度为 255
19             canvas.drawRect(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, paint);
20             if(currentLogo==null)return;                   //进行平面贴图
21             paint.setAlpha(currentAlpha);                  //设置图片透明度
22             canvas.drawBitmap(currentLogo, currentX, currentY, paint);
23         }
24         public void surfaceChanged(SurfaceHolder arg0, int arg1, int arg2, int arg3){ }
25         public void surfaceCreated(SurfaceHolder holder){//创建时被调用
26             new Thread(){                                  //线程内部类
27                 public void run(){
28                     for(Bitmap bm:logos){                  //遍历欢迎界面图片
29                         currentLogo=bm;                    //当前图片的引用
30                         currentX=SCREEN_WIDTH/2-bm.getWidth()/2;//图片位置
31                         currentY=SCREEN_HEIGHT/2-bm.getHeight()/2;
32                         for(int i=255;i>-10;i=i-10){//动态更改图片的透明度值并不断重绘
33                             currentAlpha=i;
34                             if(currentAlpha<0){            //如果当前不透明度小于零
35                                 currentAlpha=0;            //将不透明度置为零
36                             }
37                             SurfaceHolder myholder=WelcomeView.this.getHolder();
38                             Canvas canvas = myholder.lockCanvas();//获取画布
39                             try{
40                                 synchronized(myholder){    //同步
41                                     onDraw(canvas);        //进行绘制
42                             } }catch(Exception e){e.printStackTrace();
43                             }finally{
44                                 if(canvas!= null){         //如果当前画布不为空
45                                     myholder.unlockCanvasAndPost(canvas);//解锁画布
46                             } }
47                             try{
48                                 if(i==255){                //若是新图片，多等待一会
49                                     Thread.sleep(1000);
50                                 }
51                                 Thread.sleep(sleepSpan);//线程休眠一段时间
52                             }catch(Exception e){e.printStackTrace();}}}
53                     activity.hd.sendEmptyMessage(1);       //发送消息，进入到主菜单界面
54                 }
55             }.start();
56         }
57         public void surfaceDestroyed(SurfaceHolder arg0){ } //销毁时被调用
58     }
```

• 第7-15行为本类的构造器，该构造器的主要作用是初始化成员变量、设置生命周期回调接口的实现者、创建画笔对象并打开抗锯齿，同时加载图片。

• 第16-23行为本类中的绘制方法，需要根据当前Alpha值绘制图片。

• 第25-56行为实现回调接口重写的surfaceCreated方法，该方法中需要根据图片的大小以及屏幕大小确定图片的起始位置，并且不断地更改图片的透明度的值，从而达到图片逐渐消失的效果。

20.5.2 2D界面父类MySFView

本小节主要介绍所有2D界面的父类MySFView，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/d2目录下的MySFView.java。

```
package com.bn.st.d2;                              //声明包名
……//此处省略了本类中导入类的代码，读者可以自行查阅随书光盘中的源代码
public class MySFView{
    int xOffset;                                   //x轴偏移量
    int yOffset;                                   //y轴偏移量
    public void onDraw(Canvas canvas){ }           //绘制方法
    public boolean onTouchEvent(MotionEvent e){ //触控回调方法
        return false;
} }
```

> **说明**
>
> 本案例中使用到了部分2D界面，如果这些界面每一个都继承MySurfaceView并实现回调接口，那么十分不利于管理。所以这里创建一个所有2D界面的父类，在绘制时只需要选择相应的子类进行绘制即可。

20.5.3 主菜单类MenuView

由于本案例中的大部分2D界面类是继承自MySFView，所以这里不再对所有的2D界面类一一介绍，这里只介绍主界面类MenuView，读者可以通过查看随书光盘中的源代码方式了解其他的2D界面类。主菜单界面的相关代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/d2目录下的MySFView.java。

```
1 package com.bn.st.d2;                                    //声明包语句
2 ……//此处省略了本类中导入类的代码，读者可以自行查阅随书光盘中的源代码
3 public class MenuView extends MySFView{
4     MyActivity activity;                                 //Activity引用
5     ……//此处省略了本案例中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
6     int curr_menuId=0;                                   //自定义的菜单按钮编号
7     public MenuView(MyActivity activity){
8         this.activity = activity;                        //初始化activity的引用
9         initBitmap();                                    //初始化图片资源
10            if(Build.VERSION.SDK_INT<Build.VERSION_CODES.FROYO){
11                activity.showDialog(2);                  //显示Android系统版本过低对话框
12            }else if(activity.getGLVersion()<2){
13                activity.showDialog(1);                  //显示手机不具有GPU对话框
14        } }
15        ……//此处省略了本案例中初始化相关数据的代码，读者可以自行查阅随书光盘中的源代码
16        ……//此处省略了加载图片的方法，读者可以自行查阅随书光盘中的源代码
17        @Override
18        public void onDraw(Canvas canvas){               //重写的绘制方法
19            super.onDraw(canvas);
20            canvas.drawColor(Color.argb(255, 0, 0, 0)); //清屏为黑色
21            canvas.drawBitmap(background,0,0, null);     //画背景
22            if(!play_flag){                              //未按下进入游戏按钮
23                canvas.drawBitmap(button_play, button_play_x, button_play_y, null);
24            }else{                                       //按下进入游戏按钮
25                canvas.drawBitmap(button_play_press,button_play_x,button_play_y,null);
26            }
27            ……//此处省略了其他按钮的绘制代码，读者可以自行查阅随书光盘中的源代码
28        }
29        ……//此处省略了本类中触控监听的方法，读者可以自行查阅随书光盘中的源代码
30    }
```

- 第7-14行为本类的构造器，在该构造器中需要获取主控制类的对象，初始化菜单界面中需要用到的图片，并判断手机的Android系统版本是否满足要求以及手机是否具有GPU。如果手机没有GPU或者系统版本不符合要求，那么需要显示相应的对话框提示用户，同时退出游戏。
- 第20-21行为清屏并且绘制游戏背景图片。
- 第22-28行为根据每个按钮是否按下的标志位绘制相应的按钮。

20.5.4　2D界面绘制类ViewForDraw

由于所有的界面均继承自MySFView，所以在绘制时只需要根据MySFView子类的对象，调用相应的onDraw方法绘制即可，代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/d2目录下的ViewForDraw.java。

```
package com.bn.st.d2;                                   //声明包名
……//此处省略了本类中导入类的代码，读者可以自行查阅随书光盘中的源代码
public class ViewForDraw extends SurfaceView implements SurfaceHolder.Callback{
    MyActivity activity;                                //主控制类的引用
    Paint paint;                                        //画笔
    boolean flag;                                       //线程中的循环标志位
    MySFView curr;                                      //当前所在的页面
    public ViewForDraw(MyActivity activity){
        super(activity);
            this.setKeepScreenOn(true);                 //设置屏幕常亮
            this.activity = activity;                   //获取主控制类的对象
            this.getHolder().addCallback(this);         //设置生命周期回调接口的实现者
            paint = new Paint();                        //创建画笔
            paint.setAntiAlias(true);                   //打开抗锯齿
        }
        public void initThread(){                       //初始化刷帧线程
```

```
17              flag=true;
18              new Thread(){{
19                      this.setName("VFD Thread");     //设置线程名称
20                  }
21                  public void run(){                   //重写的 run 方法
22                      while(flag){
23                          repaint();                   //游戏刷帧
24                          try{Thread.sleep(40);} catch(InterruptedException e){e.
                            printStackTrace();}}}
25          }.start();}
26          public void onDraw(Canvas canvas){           //重写的绘制方法
27              if(canvas==null){                        //如果画布对象为空，则直接返回
28                  return;
29              }
30              canvas.clipRect(          //设置遮罩区域
31                  new Rect(             //规定的矩形遮罩区域
32                      0, 0,
33                      (int)com.bn.st.xc.Constant.SCREEN_WIDTH,
34                      (int)com.bn.st.xc.Constant.SCREEN_HEIGHT
35              ) );
36              curr.onDraw(canvas);      //调用相应 2D 界面中的 onDraw 方法绘制
37          }
38          public void surfaceChanged(SurfaceHolder arg0, int arg1, int arg2, int arg3){ }
39          public void surfaceCreated(SurfaceHolder holder){repaint();} //创建时被调用
40          public void surfaceDestroyed(SurfaceHolder arg0){}  //销毁时被调用
41          public void repaint(){
42              SurfaceHolder holder=this.getHolder();
43              Canvas canvas = holder.lockCanvas();    //画布加锁
44              try{
45                  synchronized(holder){ //同步绘制
46                      onDraw(canvas);
47              } }catch(Exception e){e.printStackTrace();  //捕获并打印异常信息
48              }finally{
49                  if(canvas != null){   //判断 canvas 对象是否为空，不为空，则为画布解锁
50                      holder.unlockCanvasAndPost(canvas);
51          } } }
52          @Override
53          public boolean onTouchEvent(MotionEvent e){ //屏幕触控事件
54              if(curr==null){           //判断 MySFView 对象是否为空，为空，则直接返回 false
55                  return false;
56              }
57              return curr.onTouchEvent(e);
58      } }
```

- 第8-15行为本类的构造器，该构造器中主要是获取主控制类的对象、设置生命周期回调接口的实现者、创建画笔同时打开抗锯齿。
- 第16-25行为初始化刷帧线程，主要是每隔一段固定的时间调用repaint方法刷帧。
- 第26-37行为重写的绘制方法，在该方法开始首先需要调用clipRect方法设置遮罩区域，这样就会只在遮罩区域绘制图形，然后通过设置的MySFView对象调用onDraw方法绘制图像。
- 第3840行为实现回调接口重写的surfaceChanged方法、surfaceCreated方法和surfaceDestroyed 方法。这3个方法分别在不同的场合下调用。

- 第41-51行为刷帧方法，在该方法中首先需要SurfaceHolder对象，并根据该对象锁住画布，然后调用onDraw方法绘制，最后判断canvas对象是否为空，如果不为空，则为画布解锁。
- 第53-58行为屏幕触控事件，该事件中首先判断MySFView对象是否为空，如果为空，则返回false，否则返回相应绘制界面的触控事件。

20.5.5　数据库工具类DBUtil

DBUtil类为该游戏中对数据库处理的工具类，其中主要封装了一些对数据库进行处理的方法，例如：打开数据、关闭数据库、创建表、插入数据、删除数据等方法。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/d2目录下的DBUtil.java。

```java
package com.bn.st.d2;                                    //声明包名
......//此处省略了本类中导入类的代码，读者可以自行查阅随书光盘中的源代码
public class DBUtil{
    static SQLiteDatabase sld;                           //声明 SQLiteDatabase 的引用
    public static void createorOpenDatabase(){           //打开或创建数据库
        try{
            sld=SQLiteDatabase.openDatabase(
            "/data/data/com.bn/mydb",                    //数据库所在路径
            null,                                        //CursorFactory
                SQLiteDatabase.OPEN_READWRITE|SQLiteDatabase.CREATE_IF_NECESSARY
           ); }catch(Exception e){e.printStackTrace();}//捕获并打印异常信息
        }
        public static void closeDatabase(){              //关闭数据库
            try{sld.close();}catch(Exception e){e.printStackTrace();}
        }
        public static void createTable(){                //创建表
            createorOpenDatabase();                      //打开或者创建数据库
            try{
                ......//此处省略了创建表的 SQL 语句代码，读者可以自行查阅随书光盘中的源代码
                sld.execSQL(jsSql);                      //调用 execSQL 方法创建表
                sld.execSQL(rcRSql);                     //调用 execSQL 方法创建表
            }catch(Exception e){e.printStackTrace();}
            closeDatabase();                             //关闭数据库
        }
        public static void insertJSDatabase(String date,String time){
                                                         //插入到计时赛数据库的方法
            createorOpenDatabase();                      //打开或者创建数据库
            try{
                String sql="insert into jsRecord values(null, '"+date+"','"+time+"');";
                sld.execSQL(sql);                        //调用 execSQL 方法在表中插入数据
            }catch(Exception e){e.printStackTrace();}
            closeDatabase();                             //关闭数据库
        }
        public static void insertRcRDatabase(String date,String time,int ranking){
                                                         //插入竞速赛数据库的方法
            createorOpenDatabase();                      //打开或者创建数据库
```

```
35            try{
36                String sql="insert into jRecord values(null, '"+date+"','"+time+"',
                  '"+ranking+"');";
37                sld.execSQL(sql);            //调用 execSQL 方法在表中插入数据
38            }catch(Exception e){e.printStackTrace();}
39            closeDatabase();                 //关闭数据库
40        }
41        public static List<String> queryDatabase(String tableName){//查询数据库
42            createorOpenDatabase();           //打开或者创建数据库
43            List<String> alist=new ArrayList<String>();//创建存储数据的集合
44            try{
45                String sql="select * from "+tableName;
46                Cursor cur=sld.rawQuery(sql, new String[]{});
47                while(cur.moveToNext()){           //循环结果集
48                    int tempCount=cur.getColumnCount();
49                    for(int i=1;i<tempCount;i++){
50                        alist.add(cur.getString(i));   //在集合中添加数据
51                }}
52                cur.close();                           //关闭结果集
53            }catch(Exception e){e.printStackTrace();}
54            closeDatabase();                           //关闭数据库
55            return alist;
56        }
57        public static void dropDatabase(){             //删除表
58            createorOpenDatabase();                    //打开或者创建数据库
59            try{
60                String sql="drop table jsRecord;";
61                String sql0="drop table jRecord;";
62                sld.execSQL(sql);                      //调用 execSQL 方法删除表
63                sld.execSQL(sql0);                     //调用 execSQL 方法删除表
64            }catch(Exception e){e.printStackTrace();}
65            closeDatabase();                           //关闭数据库
66        }
67        public static List<String> getTimeFromJSDatabase(){//查询计时模式表中的用时
68            createorOpenDatabase();                    //打开或者创建数据库
69            List<String> alist=new ArrayList<String>();//创建存储数据的集合
70            try{
71                String sql="select time from jsRecord;";
72                Cursor cur=sld.rawQuery(sql, new String[]{});
73                while(cur.moveToNext()){               //遍历结果集
74                    alist.add(cur.getString(0));
75                }
76                cur.close();                           //关闭结果集
77            }catch(Exception e){e.printStackTrace();}  //捕获并打印异常信息
78            closeDatabase();                           //关闭数据库
79            return alist;
80  }}
```

- 第5-12行为创建通过SQLiteDatabase调用openDatabase方法打开或者创建数据库的方法。如果没有创建数据库，则创建数据库，否则打开数据库。
- 第13-15行为创建关闭数据库的方法，主要是通过SQLiteDatabase对象调用close方法。
- 第16-24行为创建表的方法，首先需要打开或者创建数据库，然后通过SQLiteDatabase对象调用execSQL方法创建表，最后关闭数据库。

- 第25-32行为将数据插入到计时赛数据库的方法，首先打开或者创建数据库，然后通过SQLiteDatabase对象调用execSQL方法在相应的数据库中插入数据，最后关闭数据库。
- 第33-40行为创建的插入竞速赛数据库的方法，首先打开或者创建数据库，然后通过SQLiteDatabase对象调用execSQL方法在相应的数据库中插入数据，最后关闭数据库。
- 第41-56行为查询数据库所有内容的方法，首先需要打开或者创建数据库，然后通过SQLiteDatabase对象调用rawQuery方法获取结果集，最后关闭结果集以及数据库。
- 第57-66行为删除表的方法，首先打开或者创建数据库，然后通过SQLiteDatabase对象调用execSQL方法删除相应的表，最后关闭数据库。
- 第67-80行为创建的查询计时模式表中用时的方法，首先需要打开或者创建数据库，然后通过SQLiteDatabase对象调用rawQuery方法获取结果集，并遍历结果集将获取的数据添加到相应的集合中，最后关闭结果集以及数据库。

20.5.6 Android系统版本对话框AndroidVersionDialog

由于本案例中的使用到的生成对话框的代码都相似，所以对于其代码开发本小节不再对其进行一一介绍，这里只介绍生成Android系统版本过低的对话框，生成其他对话框的代码读者可以查阅随书光盘中的源代码。生成Android系统版本过低的对话框的相关代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/d2目录下的AndroidVersionDialog.java。

```
package com.bn.st.d2;                                //声明包名
......//此处省略了本类中导入类的代码，读者可以自行查阅随书光盘中的源代码
public class AndroidVersionDialog extends Dialog{    //创建继承Dialog的Android版本号类
    public AndroidVersionDialog(Context context){
      super(context,R.style.FullHeightDialog);   //调用父类
   }
    @Override
    public void onCreate (Bundle savedInstanceState){
        this.setContentView(R.layout.currandroidversion); //设置对话框显示内容
        }
        @Override
        public String toString(){                       //重写的转换为字符串的方法
            return "AndroidVersionDialog";
    } }
```

提示

创建自定义对话框时首先需要创建继承Dialog的类，然后重写相应的方法。其布局是通过xml生成的，通过重写的onCreate方法中调用setContentView方法可以设置显示界面。

20.6　选船界面相关类

前一小节已经介绍了2D界面相关类，本小节将介绍与选船界面有关的类，本节的介绍过程是 根据笔者开发选船界面的过程为读者介绍的。首先开发选船界面的各组成部分，然后开发游戏整体界面，从而逐步完成选船界面的开发。

20.6.1　着色器管理类ShaderManager

本类为着色器管理类，该类的主要作用是将着色器集中进行加载以及编译，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/xc目录下的ShaderManager.java。

```
1 package com.bn.st.xc;                                    //声明包名
2 ……//此处省略了本类中导入类的代码，读者可以自行查阅随书光盘中的源代码
3 public class ShaderManager{
4     final static int shaderCount=2;                      //着色器的数量
5     final static String[][] shaderName={                 //存储着色器的名称
6         {"vertex_tex.sh","frag_tex.sh"},{"vertex_color.sh","frag_color.sh"},
7     };
8     static String[]mVertexShader=new String[shaderCount];//存储加载的顶点着色器的数组
9     static String[]mFragmentShader=new String[shaderCount];// 存储加载的片元着色器的数组
10    static int[] program=new int[shaderCount];  //存储编译的着色器的数组
11    public static void loadCodeFromFile(Resources r){
12            for(int i=0;i<shaderCount;i++){
13             mVertexShader[i]=ShaderUtil.loadFromAssetsFile(shaderName[i][0],r);
                                                          //加载顶点着色器
14             mFragmentShader[i]=ShaderUtil.loadFromAssetsFile(shaderName[i][1], r);
                                                          //加载片元着色
15        } }
16        public static void compileShaderReal(){        //编译 shader
17            for(int i=0;i<shaderCount;i++){
18                program[i]=ShaderUtil.createProgram(mVertexShader[i], mFragment
                  Shader[i]);
19        } }
20        public static int getCommTextureShaderProgram(){//这里返回的是纹理的 shader 程序
21            return program[0];
22        }
23        public static int getColorshaderProgram(){       //这里返回的是颜色的 shader 程序
24            return program[1];
25    } }
```

- 第4行为声明的着色器的数量。第5-7行为声明的存储着色器名称的数组。第8-10行为声明的存储加载顶点着色器、片元着色器以及编译着色器的数组。
- 第11-15行为循环遍历着色器数组，通过ShaderUtil调用loadFromAssetsFile方法加载顶点着色器与片元着色器。
- 第16-19行为循环遍历着色器数组，通过ShaderUtil调用createProgram方法编译着色器。第20-22行为返回纹理着色器程序的引用。第23-25行为返回颜色着色器程序的引用。

20.6.2　围墙类ColorLightRect

ColorLightRect类为围墙类，主要是在构建选船房间时使用。该围墙的创建并未贴纹理，使用的是颜色与光照，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/xc目录下的ColorLightRect.java。

```java
package com.bn.st.xc;
……//此处省略了本类中导入类的代码，读者可以自行查阅随书光盘中的源代码
public class ColorLightRect{
    int mProgram;                                    //自定义渲染管线程序
    ……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
   int vCount=0;                                     //顶点数量
   public ColorLightRect(int mProgramIn,float width,float height,float []color){
           this.r=color[0]; this.g=color[1]; this.b=color[2];
           initVertexData(width,height);                   //初始化顶点数据
             initShader(mProgramIn);                 //初始化着色器
       }
       public void initVertexData(float width,float height){//初始化顶点数据的方法
           vCount=6;                                 //声明顶点数量
           float vertices[]=new float[]{            //矩形各个顶点坐标值
             -width,height,0,  -width,-height,0,  width,-height,0,
             width,-height,0,  width,height,0,  -width,height,0
           };
           float normals[]=new float[]{             //矩形各个顶点的颜色值
             0,0,1,  0,0,1,  0,0,1,  0,0,1,  0,0,1,  0,0,1,
           };
           ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
           vbb.order(ByteOrder.nativeOrder());  //设置字节顺序
           mVertexBuffer = vbb.asFloatBuffer(); //转换为 Float 型缓冲
           mVertexBuffer.put(vertices);             //向缓冲区中放入顶点坐标数据
           mVertexBuffer.position(0);               //设置缓冲区起始位置
           ByteBuffer nbb = ByteBuffer.allocateDirect(normals.length*4);
           nbb.order(ByteOrder.nativeOrder());  //设置字节顺序
           mNormalBuffer = nbb.asFloatBuffer(); //转换为 Float 型缓冲
           mNormalBuffer.put(normals);              //向缓冲区中放入顶点着色数据
           mNormalBuffer.position(0);               //设置缓冲区起始位置
       }
       public void initShader(int mProgramIn){ //初始化着色器的 initShader 方法
           mProgram = mProgramIn;                   //获取自定义渲染管线程序引用
           maPositionHandle = GLES20.glGetAttribLocation(mProgram, "aPosition");
           maNormalHandle= GLES20.glGetAttribLocation(mProgram, "aNormal");
           muMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
           muMMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMMatrix");
           maLightLocationHandle=GLES20.glGetUniformLocation(mProgram, "uLightLocation");
           maCameraHandle=GLES20.glGetUniformLocation(mProgram, "uCamera");
           maColorR=GLES20.glGetUniformLocation(mProgram, "colorR");
           maColorG=GLES20.glGetUniformLocation(mProgram, "colorG");
           maColorB=GLES20.glGetUniformLocation(mProgram, "colorB");
           maColorA=GLES20.glGetUniformLocation(mProgram, "colorA");
       }
       public void drawSelf(float Alpha){
           GLES20.glUseProgram(mProgram);  //指定使用某套着色器程序
           GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, false, MatrixState.getFinalMatrix(), 0);
           GLES20.glUniformMatrix4fv(muMMatrixHandle, 1, false, MatrixState.getMMatrix(), 0);
```

```
49          GLES20.glUniform3fv(maLightLocationHandle, 1, MatrixState.lightPositionFB);
50           GLES20.glUniform3fv(maCameraHandle, 1, MatrixState.cameraFB);
51          GLES20.glVertexAttribPointer(maPositionHandle,3,GLES20.GL_FLOAT,false,3*4,
            mVertexBuffer);
52           GLES20.glVertexAttribPointer(maNormalHandle,3,GLES20.GL_FLOAT,false,3*4,
             mNormalBuffer);
53           GLES20.glEnableVertexAttribArray(maPositionHandle);
54           GLES20.glEnableVertexAttribArray(maNormalHandle);
55           GLES20.glUniform1f(maColorR , r);        //向着色器中传入颜色的红色通道值
56           GLES20.glUniform1f(maColorG , g);        //向着色器中传入颜色的绿色通道值
57           GLES20.glUniform1f(maColorB , b);        //向着色器中传入颜色的蓝色通道值
58           GLES20.glUniform1f(maColorA , Alpha);    //向着色器中传入颜色的透明度通道值
59           GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount);
60      } }
```

- 第4-6行为本类中成员变量的声明。第7-11行为本类的构造器，主要包括初始化颜色值、初始化顶点数据以及初始化着色器程序。
- 第12-31行为初始化顶点坐标与着色数据的方法，在该方法中需要指定矩形的顶点坐标，将其存储到缓冲区中，并设置缓冲区的起始位置。指定纹理矩形顶点的颜色，将颜色数组存储到缓冲区中，同样也需要设置缓冲区的起始位置。
- 第32-44行为初始化着色器的方法，该方法中首先需要获取自定义渲染管线程序引用，然后根据着色器中各个值的属性调用GLES20相应的方法获取属性的引用。
- 第45-60行为绘制方法，首先需要指定使用某套着色器程序，然后向着色器中传入相应的数据，最后通过GLES20调用glDrawArrays方法绘制图形。

20.6.3　选船房间类HouseForDraw

HouseForDraw类为选船房间类，该选船房间主要是由地面、带纹理的围墙以及带颜色的围墙组成，代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/xc目录下的HouseForDraw.java。

```
package com.bn.st.xc;                                      //声明包名
......//此处省略了本类中导入类的代码，读者可以自行查阅随书光盘中的源代码
public class HouseForDraw{                                 //此类主要是用来绘制房间
    private float floorWidth=60    ;                       //地面的长、宽以及向下平移的距离
    ......//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
    private float alpha1=1.0f;                             //透明度值 1
    private float alpha2=0.3f;                             //透明度的值
    public HouseForDraw(){
        floor=new TextureRect(ShaderManager.getCommTextureShaderProgram(),
                floorWidth,floorHeight);                   //创建地板类
            wall=new ColorLightRect(ShaderManager.getColorshaderProgram(),
                wallWidth,wallHeight,HOUSE_COLOR[1]);
            wall_tex=new TextureRect(ShaderManager.getCommTextureShaderProgram(),
                wallWidth,wallHeight);                     //创建纹理矩形类
        }
        public void drawFloor(int texId){                  //绘制地板的方法
```

```java
            MatrixState.pushMatrix();                     //保护现场
            MatrixState.translate(0, floorDownOffset,0);
            MatrixState.rotate(-90, 1, 0, 0);
            floor.drawSelf(texId);                        //绘制地板
            MatrixState.popMatrix();                      //恢复矩阵
        }
        public void drawSelf(){                           //绘制不透明的方法
            MatrixState.pushMatrix();                     //保护现场
            MatrixState.translate(0, 0,-wall_z_offset);
            wall.drawSelf(alpha1);                        //绘制围墙 1
            MatrixState.popMatrix();                      //恢复现场
            MatrixState.pushMatrix();                     //保护现场
            MatrixState.rotate(120, 0,1, 0);
            MatrixState.translate(0,0, -wall_z_offset);
            wall.drawSelf(alpha1);                        //绘制围墙 3
            MatrixState.popMatrix();                      //恢复现场
            MatrixState.pushMatrix();                     //保护现场
            MatrixState.rotate(240, 0,1, 0);
            MatrixState.translate(0,0, -wall_z_offset);
            wall.drawSelf(alpha1);                        //绘制围墙 5
            MatrixState.popMatrix();                      //恢复现场
        }
        public void drawTexWall(int[] texId,int index){ //绘制广告墙的方法
            MatrixState.pushMatrix();                     //保护现场
            MatrixState.rotate(60, 0,1, 0);
            MatrixState.translate(0,0, -wall_z_offset);
            wall_tex.drawSelf(texId[index]);              //绘制围墙 2
            MatrixState.popMatrix();                      //恢复现场
            MatrixState.pushMatrix();                     //保护现场
            MatrixState.rotate(180, 0,1, 0);
            MatrixState.translate(0,0, -wall_z_offset);
            wall_tex.drawSelf(texId[index]);              //绘制围墙 4
            MatrixState.popMatrix();                      //恢复现场
            MatrixState.pushMatrix();                     //保护现场
            MatrixState.rotate(300, 0,1, 0);
            MatrixState.translate(0,0, -wall_z_offset);
            wall_tex.drawSelf(texId[index]);              //绘制围墙 6
            MatrixState.popMatrix();                      //恢复现场
        }
        public void drawTransparentWall(){                //绘制透明围墙的方法
            MatrixState.pushMatrix();                     //保护现场
            MatrixState.translate(0, 0,-wall_z_offset+0.05f);
            wall.drawSelf(alpha2);                        //绘制围墙 1
            MatrixState.popMatrix();                      //恢复现场
            MatrixState.pushMatrix();                     //保护现场
            MatrixState.rotate(120, 0,1, 0);
            MatrixState.translate(0,0, -wall_z_offset+0.05f);
            wall.drawSelf(alpha2);                        //绘制围墙 3
            MatrixState.popMatrix();                      //恢复现场
```

```
66            MatrixState.pushMatrix();                //保护现场
67            MatrixState.rotate(240, 0,1, 0);
68            MatrixState.translate(0,0, -wall_z_offset+0.05f);
69            wall.drawSelf(alpha2);                   //绘制围墙5
70            MatrixState.popMatrix();                 //恢复现场
71    } }
```

- 第8-15行为本类的构造器，在该构造器中主要是创建地板类对象、带纹理的矩形对象以及不带纹理的矩形对象。
- 第16-22行为绘制地板的方法，首先需要保护现场，然后将地板进行平移以及旋转，最后绘制地板并恢复现场。
- 第23-38行为绘制选船房间中的不透明围墙。第39-55行为绘制选船房间中的广告围墙。第56-71行为绘制选船房间中的透明围墙。

20.6.4　展台类DisplayStation

DisplayStation类为展台类，主要是绘制展台自身的圆柱以及展台上的圆面，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/xc目录下的DisplayStation.java。

```
1package com.bn.st.xc;                                        //声明包名
2……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
3public class DisplayStation{                                 //绘制展台类
4     CylinderTextureByVertex cylinderStation;               //展台圆柱
5     private float aspan=8;                                  //水平切分份数
6     private float lspan=1;                                  //竖直切分份数
7     private float length;
8     CircleForDraw circle;                                   //展台的圆面
9     private float alpha1=1.0f;                              //展台的透明度值 1
10        private float alpha2=0.5f;                          //展台的透明度值 2
11        public DisplayStation(float radius,float length){
12            this.length=length;                             //圆柱高度
13            cylinderStation=new CylinderTextureByVertex(ShaderManager.
              getColorshaderProgram(),
14                    radius,length,aspan,lspan,COLOR_CYLINDER);
15            circle=new CircleForDraw(ShaderManager.getColorshaderProgram(
16                    aspan, radius,COLOR_CIRCLE);            //创建圆面对象
17        }
18        public void drawSelfCylinder(){                     //绘制自身圆柱面
19            GLES20.glDisable(GLES20.GL_CULL_FACE);          //关闭背面剪裁
20            cylinderStation.drawSelf(alpha1);               //绘制圆柱
21            GLES20.glEnable(GLES20.GL_CULL_FACE);           //打开背面剪裁
22        }
23        public void drawSelfCircle(){                       //绘制自身的圆面
24            MatrixState.pushMatrix();                       //保护现场
25            MatrixState.translate(0, length/2, 0);
26            MatrixState.rotate(-90, 1, 0, 0);
27            circle.drawSelf(alpha1);                        //绘制圆面
28            MatrixState.popMatrix();                        //恢复现场
29        }
30        public void drawTransparentCircle(){                //画透明的圆面
31            MatrixState.pushMatrix();                       //保护现场
32            MatrixState.translate(0, -Constant.HOUSE_GAO/2+length/2,0);
33            MatrixState.rotate(-90, 1, 0, 0);
34            circle.drawSelf(alpha2);                        //绘制透明圆面
35            MatrixState.popMatrix();                        //恢复现场
36    } }
```

- 第4-10行为本类成员变量的声明，包括圆柱的引用以及其相关属性、圆面的引用以及圆面的两个透明度的值。
- 第11-17行为本类的构造器，在该构造器中主要是获取圆柱的高度，创建圆柱对象与圆面对象。
- 第18-22行为绘制自身圆柱面的方法，在绘制圆柱面之前需要关闭背面剪裁，绘制完毕后再打开背面剪裁。
- 第23-29行为绘制自身圆面的方法。第30-36行为绘制透明圆面的方法。

20.6.5 赛艇类Boat

本游戏中的赛艇模型是由多个obj文件构成的，因此需要开发一个赛艇类，用于将赛艇的所有部件组合并绘制，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/xc目录下的Boat.java。

```
1 package com.bn.st.xc;                                    //声明包名
2 ……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
3 public class Boat{                                       //赛艇类
4     List<LoadedObjectVertexTexXC> lovo=new ArrayList<LoadedObjectVertexTexXC>();
5     public Boat(LoadedObjectVertexTexXC[] parts,XCSurfaceView mv){
6         for(int i=0;i<parts.length;i++){                 //遍历加载的赛艇部件的列表
7             lovo.add(parts[i]);                          //将加载的部件添加到相应的集合中
8             parts[i].initShader(ShaderManager.getCommTextureShaderProgram());
9     } }
10        public void drawSelf(int[] texId){               //绘制赛艇的方法
11            for(int i=0;i<lovo.size();i++){              //循环遍历集合，绘制赛艇每一部分
12                lovo.get(i).drawSelf(texId[i]);
13    } } }
```

- 第4行为本类的成员变量，主要是存储需要加载赛艇的每个部件。
- 第5-9行为本类的构造器，该构造器的主要作用是将加载的赛艇部件添加到集合中，同时初始化赛艇各个部件的着色器。
- 第10-13行为绘制赛艇的方法，主要是遍历存储赛艇部件的集合，然后调用相应的drawSelf方法绘制各个部件，最后就生成一个赛艇。

20.6.6 选船界面XCSurfaceView

本小节主要介绍选船界面XCSurfaceView的开发，该类继承自GLSurfaceView。同时需要在该类中创建渲染内部类，在该渲染类中对选船界面进行开发，具体开发步骤如下。

（1）首先开发XCSurfaceView类的框架，理解该框架有助于读者对于本类的整体理解，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/xc目录下的XCSurfaceView.java。

```java
package com.bn.st.xc;                                    //声明包名
......//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
public class XCSurfaceView extends GLSurfaceView{
    final float TOUCH_SCALE_FACTOR = 180.0f/SCREEN_WIDTH;//角度缩放比例
    ......//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
    public boolean flag_display=true;                    //广告标志位
    public XCSurfaceView(Context context){               //本类的构造器
      super(context);
      this.setEGLContextClientVersion(2);                //设置使用 OpenGL ES2.0
          mRenderer = new SceneRenderer();               //创建场景渲染器
          setRenderer(mRenderer);                        //设置渲染器
          setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);//设置渲染模式
          this.setKeepScreenOn(true);                    //保持屏幕常亮
          cx=(float)(tx+                                 //摄像机 x 坐标
            Math.cos(Math.toRadians(xAngle))*Math.sin(Math.toRadians(yAngle))
            *XC_DISTANCE);
          cz=(float)(tz+                                 //摄像机 y 坐标
            Math.cos(Math.toRadians(xAngle))*Math.cos(Math.toRadians(yAngle))
            *XC_DISTANCE);
          cy=(float)(ty+Math.sin(Math.toRadians(xAngle))*XC_DISTANCE);//摄像机 y 坐标
      }
        ......//此处省略了触摸事件回调的方法，将在下面进行介绍
        ......//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
        static LoadedObjectVertexNormalXC romeData; //赛艇部件加载类的引用
        public static void loadVertexFromObj(Resources r){   //加载赛艇部件的方法
            romeData=LoadUtilXC.loadFromFileVertexOnly("rome.obj",r, 1f, 1f, 1f);
            for(int j=0;j<BoatInfo.boatPartNames.length;j++){
                for(int i=0;i<BoatInfo.boatPartNames[j].length;i++){
                    parts[j][i]=LoadUtilTexXC.loadFromFileVertexOnly(
                        BoatInfo.boatPartNames[j][i],r,
                        ShaderManager.getCommTextureShaderProgram()
        ); } } }
        private class SceneRenderer implements GLSurfaceView.Renderer{
          public void onDrawFrame(GL10 gl){              //重写的 onDrawFrame 方法
            GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT | GLES20.GL_COLOR_BUFFER_BIT);
             MatrixState.setProjectFrustum(-ratio, ratio, -1, 1, 2.5f, 1000);
             MatrixState.setCamera(cx,cy,cz,tx,ty,tz,upX,upY,upZ);
             MatrixState.copyMVMatrix();                 //复制总变换矩阵
             MatrixState.setLightLocation(cx, cy+5, cz+3);//初始化光源位置
             drawBoatMirrorOnWall();                     //绘制赛艇在墙面的影子
             GLES20.glEnable(GLES20.GL_BLEND);           //开启混合
            GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
             house.drawTransparentWall();
             GLES20.glDisable(GLES20.GL_BLEND);          //关闭混合
             house.drawFloor(texFloorId);                //绘制地板
```

```
44              house.drawTexWall(texWallId,tex_index);    //绘制广告墙
45              MatrixState.pushMatrix();                  //保护现场
46              MatrixState.translate(0, -Constant.HOUSE_GAO/2, 0);
47              displayStation.drawSelfCylinder();         //绘制展台的不透明的圆柱
48              MatrixState.popMatrix();                   //恢复现场
49              MatrixState.pushMatrix();                  //保护现场
50              drawBoatShadow();                          //绘制船在展台上的倒影
51              MatrixState.popMatrix();                   //恢复现场
52              GLES20.glEnable(GLES20.GL_BLEND);          //开启混合
53              GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
54              displayStation.drawTransparentCircle();
55              GLES20.glDisable(GLES20.GL_BLEND);         //关闭混合
56              drawBoat();                                //绘制真实船
57              drawRomeColumn();                          //绘制柱子
58              drawButton();                              //绘制上一艘、下一艘按钮
59          }
60          ……//此处省略了重写的 onSurfaceChanged 方法，将在下面给出
61          ……//此处省略了重写的 onSurfaceCreated 方法，读者可以自行查阅随书光盘中的源代码
62          ……//此处省略了绘制柱子的方法，读者可以自行查阅随书光盘中的源代码
63          ……//此处省略了绘制船以及船倒影的方法，将在下面给出
64          ……//此处省略了绘制相关按钮的方法，读者可以自行查阅随书光盘中的源代码
65      }
66      ……//此处省略了虚拟按钮的监听方法的方法，读者可以自行查阅随书光盘中的源代码
67      ……//此处省略了生成纹理 id 的方法，将在下面介绍
68      ……//此处省略了加载 Bitmap 位图的方法，将在下面介绍
69  }
```

- 第7-19行为本类的构造器，在该构造器中主要是设置使用OpenGL ES 2.0、创建渲染器、设置渲染器为自定义的渲染器、设置渲染模式为主动渲染以及计算摄像机的坐标。
- 第23-30行为加载赛艇部件的方法，首先需要遍历BoatInfo类中的每个赛艇具有的部件数量，然后通过LoadUtilTexXC调用loadFromFileVertexOnly方法加载各个部件。
- 第32-59行为实现GLSurfaceView.Renderer接口后重写的onDrawFrame方法，在该方法中绘制选船界面中的所有可见模块。

（2）接下来开发本类的触控事件，该方法中会监听玩家在触控区域的移动，并根据玩家在触控区域横向的运动动态更改摄像机的位置，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/xc目录下的XCSurfaceView.java。

```
@Override
public boolean onTouchEvent(MotionEvent e){      //重写的触控方法
    float y = e.getY();                          //得到按下点 Y 坐标
    float x = e.getX();                          //得到按下点 X 坐标
    float domain_span=5;
    switch (e.getAction()){
        case MotionEvent.ACTION_MOVE:
            float dx = x - mPreviousX;           //计算触控点 X 位移
            float dy = y- mPreviousY;            //计算触控点 Y 位移
```

```
                yAngle -= dx * TOUCH_SCALE_FACTOR;
                xAngle -= dy * TOUCH_SCALE_FACTOR/2.0f;//仰角改变
                if(xAngle>45){                      //仰角的最大值
                    xAngle=45;
                }
                if(xAngle<10){                      //仰角的最小值
                    xAngle=10;
                }
                if(Math.abs(dx)>domain_span){       //确定一个阈值
                 isMoved=true;                      //表示当前移动过
                 curr_angle_speed=ori_angle_speed*(-dx/SCREEN_WIDTH);//设置当前角速度
                  if(dx>20||dx<-20){                //判断触控点 X 位移是否大于 20
                      if(curr_angle_speed>0){
                          curr_angle_speed=curr_angle_speed+ori_angle_speed;
                       }else{
                          curr_angle_speed=curr_angle_speed-ori_angle_speed;
                   } }
                  acceleratedVelocity=0.03f;        //设置加速度大小
                  if(curr_angle_speed>0){           //若当前角速度大于零则加速度置为负
                      acceleratedVelocity=-0.03f;
             } }
             cx=(float)(tx+                         //重新计算摄像机的坐标
               Math.cos(Math.toRadians(xAngle))*Math.sin(Math.toRadians(yAngle))
               *XC_DISTANCE);
             cz=(float)(tz+
               Math.cos(Math.toRadians(xAngle))*Math.cos(Math.toRadians(yAngle))
               *XC_DISTANCE);
             cy=(float)(ty+Math.sin(Math.toRadians(xAngle))*XC_DISTANCE);
          break;
          case MotionEvent.ACTION_DOWN :            //触控事件为按下
               flag_go=false;                       //设置允许惯性标志位为 false
               if(x>button1[0]&&x<button1[1]&&y>button1[2]&&y<button1[3]){
                   index_boat=(index_boat-1+3)%3;
               }
               if(x>button2[0]&&x<button2[1]&&y>button2[2]&&y<button2[3]){
                   index_boat=(index_boat+1)%3;
               }
               BoatInfo.cuttBoatIndex=index_boat;//获取当前赛艇的索引值
           break;
          case MotionEvent.ACTION_UP:               //触控事件为抬起
               if(isMoved){                         //判断是否移动
                   flag_go=true;                    //设置惯性标志位为 true
                   countGoStep=countGoStepInitValue;
                   isMoved=false;                   //设置移动标志位为 false
               }
           break;
       }
      mPreviousX = x;                               //记录触控点 X 位置
      mPreviousY=y;                                 //记录触控点 Y 位置
      cx=(float)(tx+Math.cos(Math.toRadians(xAngle))*Math.sin(Math.toRadians
      (yAngle))*XC_DISTANCE);
      cz=(float)(tz+Math.cos(Math.toRadians(xAngle))*Math.cos(Math.toRadians
      (yAngle))*XC_DISTANCE);
      cy=(float)(ty+Math.sin(Math.toRadians(xAngle))*XC_DISTANCE);//摄像机 y 坐标
      return true;
  }
```

- 第3-5行为获取触控点的坐标以及移动的阀值。
- 第7-36行为判断触控事件为移动时，首先判断触控点的偏移量，并根据偏移量计算仰角以及偏移角的变化，然后计算curr_angle_speed的值，最后计算摄像机的坐标。
- 第37-46行为触控事件为按下时，首先设置允许惯性标志位为false，然后判断是否按到上一艘或者下一艘按钮，并获取当前赛艇的索引值。
- 第47-53行为触控事件为抬起时，需要设置惯性标志位为true，并将移动标志位设置为false。
- 第55-60行为记录上一触控点的*x* 位置以及*y* 位置，并计算摄像机的坐标。

（3）接下来开发onSurfaceChanged方法，该方法为实现GLSurfaceView.Renderer接口重写的。其在界面改变时调用，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/xc目录下的XCSurfaceView.java。

```
public void onSurfaceChanged(GL10 gl, int width, int height){
    GLES20.glViewport(0, 0, width, height);     //设置视窗大小及位置
    ratio = (float) width / height;              //计算 GLSurfaceView 的宽高比
    virtualButton();                             //虚拟按钮的监听方法
    new Thread(){                                //匿名内部类
        @Override
        public void run(){                       //重写的 run 方法
            while(flagForThread){
            try{
                if(flag_go){                     //如果允许惯性
                    countGoStep--;
                    if(countGoStep<=0){
                        curr_angle_speed=curr_angle_speed+acceleratedVelocity;
                    }
                    if(Math.abs(curr_angle_speed)>0.1f){
                        yAngle=yAngle+curr_angle_speed;
                        cx=(float)(tx+Math.cos(Math.toRadians(xAngle))*
                            Math.sin(Math.toRadians(yAngle))*XC_DISTANCE);
                        cz=(float)(tz+Math.cos(Math.toRadians(xAngle))*
                           Math.cos(Math.toRadians(yAngle))*XC_DISTANCE);
                        cy=(float)(ty+Math.sin(Math.toRadians(xAngle))*XC_DISTANCE);
                    }else{
                        curr_angle_speed=0;//计算当前的角速度
                        flag_go=false;    //设置允许惯性标志位为 false
                    } }
                        Thread.sleep(10);     //线程休眠 10 ms
                    }catch (InterruptedException e){e.printStackTrace();}
        } } }.start();
    }
```

- 第2-4行为设置视窗大小及位置、计算GLSurfaceView的宽高比并调用virtualButton方法为虚拟按钮添加监听。
- 第5-28行为线程匿名内部类，在该内部类中需要根据curr_angle_speed值计算yAngle变化以及摄像机的位置坐标。如果curr_angle_speed值大于0.1则计算，否则设置curr_angle_speed值为0，并且设置允许惯性标志位为false。

（4）接下来将开发选船界面中赛艇的绘制以及赛艇在展台以及四周墙壁倒影的绘制的相关代码，其实3D世界中，倒影的绘制只是将物体移动到该物体倒影应该在的位置再绘制一遍。本案例中绘制赛艇以及倒影的代码如下。

代码位置：见随书光盘中源代码/第20

章/BNST/com/bn/st/xc目录下的XCSurfaceView.java。

```
1 public void drawBoat(){                                  //绘制船的方法
2      MatrixState.pushMatrix();                           //保护现场
3     switch(index_boat){                                  //判断赛艇的索引值
4          case 0:
5              MatrixState.translate(0, -10f, 0);          //设置赛艇沿y轴偏移10个单位
6              MatrixState.scale(RATIO_BOAT, RATIO_BOAT, RATIO_BOAT);
7              boat[0].drawSelf(heroBoatTexId);            //绘制1号赛艇
8          break;
9          ……//此处省略了其他赛艇的绘制，读者可以自行查阅随书光盘中的源代码
10         }
11         MatrixState.popMatrix();
12    }
13    public void drawBoatShadow(){                        //绘制船的倒影的方法
14         GLES20.glDisable(GLES20.GL_CULL_FACE);          //关闭背面剪裁
15         MatrixState.pushMatrix();                       //保护现场
16         switch(index_boat){                             //判断赛艇的索引值
17             case 0:
18                 MatrixState.translate(-1f, -14f, 0); //设置赛艇的平移矩阵
19                 MatrixState.rotate(180, 0, 0, 1);
20                 MatrixState.scale(RATIO_BOAT, RATIO_BOAT, RATIO_BOAT);
21                 boat[0].drawSelf(heroBoatTexId);    //绘制1号赛艇
22             break;
23             ……//此处省略了其他赛艇倒影的绘制，读者可以自行查阅随书光盘中的源代码
24        }
25        MatrixState.popMatrix();
26        GLES20.glEnable(GLES20.GL_CULL_FACE);            //打开背面剪裁
27    }
28    public void drawBoatMirrorOnWall(){                  //绘制船在墙中的镜像的方法
29         final float ydistance=6f;                       //赛艇y轴偏移量
30         final float zdistance=-43f;                     //赛艇z轴偏移量
31         int k=(int) Math.abs(yAngle)/360+2;
32         float yAngleTemp=(yAngle+360*k)%360;
33         final int span=68;
34         if(yAngleTemp<=span&&yAngleTemp>=0||yAngleTemp>=360-span&&yAngleTemp<=360){
35             MatrixState.pushMatrix();                   //保护现场
36             MatrixState.translate(0, ydistance, zdistance);
37             MatrixState.rotate(180, 0, 1, 0);           //设置赛艇沿y轴旋转180°
38             MatrixState.rotate(-15, 1, 0, 0);
39             drawBoat();                                 //绘制赛艇
40             MatrixState.popMatrix();                    //恢复现场
41         }
42         ……//此处省略了赛艇在其他面中影子的绘制，读者可以自行查阅随书光盘中的源代码
43    }
```

- 第1-12行为绘制赛艇的方法，首先需要保护现场，然后根据index_boat的值选择相应的赛艇绘制，最后恢复现场。
- 第13-27行绘制赛艇在展台上倒影的方法，首先需要关闭背面剪裁并保护现场，然后根据index_boat的值选择相应的赛艇，并将赛艇旋转到相应倒影的位置并绘制，最后恢复现场。
- 第28-43行为绘制船在墙壁中镜像的方法，由于本选船界面中有3面墙壁是存在镜像的，因此需要分3种情况绘制镜像。其中绘制镜像与绘制倒影相似，均是将赛艇移动到其镜像位置，并绘制。

（5）接下来开发游戏中根据Bitmap位图文件获取系统分配的纹理id的方法，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/xc目录下的XCSurfaceView.java。

```
1 public int initTextureFromBitmap(Bitmap bitmapTmp){
2     int[] textures = new int[1];                    //存储纹理 id 的数组
3     GLES20.glGenTextures(
4             1,                                      //产生的纹理 id 的数量
5             textures,                               //纹理 id 的数组
6             0                                       //偏移量
7     );
8     int textureId=textures[0];                      //将分配的纹理 id 存储到临时变量中
9     GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureId);//绑定纹理
10        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
11                    GLES20.GL_TEXTURE_MIN_FILTER,GLES20.GL_NEAREST);
12        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
13                    GLES20.GL_TEXTURE_MAG_FILTER,GLES20.GL_NEAREST);
14        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
15                    GLES20.GL_TEXTURE_WRAP_S,GLES20.GL_REPEAT);
16        GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
17                    GLES20.GL_TEXTURE_WRAP_T,GLES20.GL_REPEAT);
18       GLUtils.texImage2D(                              //实际加载纹理
19            GLES20.GL_TEXTURE_2D, //纹理类型，在 OpenGL ES 中必须为 GL10.GL_TEXTURE_2D
20            0,                    //纹理的层次，0 表示基本图像层，可以理解为直接贴图
21            bitmapTmp,            //纹理图像
22            0                     //纹理边框尺寸
23       );
24       return textureId;          //返回系统分配的纹理 id
25    }
```

- 第2行为声明的存储纹理id的数组，在OpenGL ES中系统分配的纹理id均是存储到数组中的。
- 第9-17行为设置纹理采样方式，有关纹理采样的详情读者可以参见本书第四章。
- 第18-24行为实际加载纹理的方法，该方法中需要指明纹理类型、设置纹理层次、指明纹理图片以及纹理边框的尺寸。

20.7　游戏界面相关类

前几节已经介绍了主控制类的开发、2D界面相关类的开发以及选船界面相关类的开发，接下 来将介绍本案例中最重要的游戏界面相关类的开发。其介绍顺序与前面介绍选船界面相关类的开发相同，首先介绍各个组成部分的开发，然后是游戏整体界面的开发，逐步完成游戏界面开发。

20.7.1 进度条类Process

本游戏在进入游戏界面之前首先进入的是加载界面，而加载界面中最难开发的就是进度条，因此本小节将对加载界面中进度条的开发进行介绍，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的Process.java。

```
package com.bn.clp;
......//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
public class Process{
    int mProgram;                                   //自定义渲染管线引用
    ......//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
    boolean flag=true;                              //循环标志位
    float currPrograss;                             //当前进度
    float width;                                    //进度条宽度
    float percent;                                  //percent 表示进度值
        public Process(int programId,float width,float height,float[] texCoor,float
        percent){
            this.width=width;                       //设置当前的宽度
            this.percent=percent;                   //设置当前的进度
            initVertexData(width,height,texCoor);//初始化顶点数据
            initShader(programId);                  //初始化 Shader 程序
            new Thread(){
                @Override
                public void run(){                  //重写的 run 方法
                    while(flag){                    //while 循环
                        if(currPrograss>=Process.this.width){
                            flag=false;             //设置循环标志位为 false
                            break;                  //退出
                        }
                        currPrograss=percentToWidth(Process.this.percent,Process.
                        this.width);
                        try{Thread.sleep(100);}catch(Exception e){e.printStack
                        Trace();}
            } } }.start();
        }
        ......//此处省略了初始化顶点数据的方法，读者可以自行查阅随书光盘中的源代码
        public void initShader(int programId){ //初始化 Shader 的方法
          mProgram =programId;                      //基于顶点着色器与片元着色器创建程序
          maPositionHandle = GLES20.glGetAttribLocation(mProgram, "aPosition");
          maTexCoorHandle= GLES20.glGetAttribLocation(mProgram, "aTexCoor");
          muMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
         maPrograss= GLES20.glGetUniformLocation(mProgram, "aPrograss");//进度的引用 id
        }
        ......//此处省略了自定义的绘制方法 drawSelf，读者可以自行查阅随书光盘中的源代码
        public float percentToWidth(float percent,float width){
                                                    //将百分比对应成与 width 相关的数据
```

```
37            return percent*width*2-width;
38      } }
```

- 第6-9行为本类中与进度条有关的部分成员变量的声明，主要包括与进度条有关的进度值以及进度条的宽度值。
- 第10-26行为本类的构造器，主要是初始化相应的成员变量，初始化顶点数据以及初始化着色器数据，同时开启线程实时计算当前进度条纹理图的坐标值。
- 第28-33行为初始化着色器的方法，首先获取自定义渲染管线程序的引用，然后通过GLES20调用相应的方法获取顶点数据的引用、顶点纹理坐标数据的引用以及进度条的引用。

20.7.2　3D物体父类BNDrawer

前一小节介绍了进度条类的开发，接下来开发3D物体的父类——DNDrawer，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的BNDrawer.java。

```
package com.bn.clp;                                          //声明包名
public abstract class BNDrawer{                              //抽象公共类
    public abstract void drawSelf(int[] texId,int dyFlag);   //绘制方法
}
```

> **说明**
>
> 通过构建3D物体的父类，可以非常容易地管理各个3D物体，同时易于添加其他3D物体。在本案例中该类的子类主要包括飞艇、半崖子、桥、城堡、船埠头、小山、广告以及隧道。

20.7.3　3D物体控制类TDObjectForControl

由于生成3D物体的代码结构都十分相似，所以这里就不再对3D物体的生成代码进行介绍，这里将直接开发3D物体控制类——TDObjectForControl，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的TDObjectForControl.java。

```
1 package com.bn.clp;                                          //声明包名
2 ……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
3 public class TDObjectForControl{                             //3D 物体控制类
4     BNDrawer bndrawer;                                       //3D 物体的引用
5     int objectId;                                            //物件的 id
6     float x; float y; float z;                               //物体的平移坐标
7     float yAngle;                                            //物体绕 y 轴旋转的角度
8     int rows; int cols;                                      //物体所在的行与列
9     public TDObjectForControl(BNDrawer bndrawer,int objectId,float x,
10                         float y,float z,float yAngle,int rows,int cols){
11            this.bndrawer=bndrawer;                          //创建 3D 物体对象
12            this.objectId=objectId;                          //获取物件 id
13            this.x=x; this.y=y; this.z=z;                    //初始化物体平移坐标
14            this.yAngle=yAngle;                              //初始化物体绕 y 轴旋转的角度
15            this.rows=rows; this.cols=cols;                  //初始化物体的行与列
16     }
17     public void drawSelf(int[] texId,int dyFlag){           //自定义的绘制方法 drawSelf
18         if(dyFlag==0){                                      //绘制实体
19             MatrixState.pushMatrix();                       //保护现场
20             MatrixState.translate(x, y, z);
21             MatrixState.rotate(yAngle, 0, 1, 0);
22             bndrawer.drawSelf(texId,dyFlag);                //绘制 3D 物体
23             MatrixState.popMatrix();                        //恢复现场
24         }else if(dyFlag==1){                                //绘制倒影
25             float yTranslate=y;                             //实际绘制时 Y 的零点
26             float yjx=(0-yTranslate)*2;                     //进行镜像绘制时的调整值
27             GLES20.glDisable(GLES20.GL_CULL_FACE);//关闭背面剪裁
28             MatrixState.pushMatrix();                       //保护现场
29             MatrixState.translate(x, y, z);
30             MatrixState.rotate(yAngle, 0, 1, 0);            //将物体绕 y 轴旋转一定的角度
31             MatrixState.translate(0, yjx, 0);
32             MatrixState.scale(1, -1, 1);
33             bndrawer.drawSelf(texId,dyFlag);                //绘制物体
34             MatrixState.popMatrix();                        //恢复现场
35             GLES20.glEnable(GLES20.GL_CULL_FACE);//打开背面剪裁
36 } } }
```

- 第4-8行为本类中成员变量的声明，主要包括3D物体的引用、物体id、物体的平移坐标、物体绕*y* 轴转动的角度以及物体所在的行与列。
- 第9-16行为本类的构造器，在该构造器中主要是初始化相应的成员变量。
- 第18-24行为绘制实体的代码，首先需要保护现场，然后将3D物体进行平移以及旋转并绘制，最后恢复现场。

• 第24-36行为绘制3D物体在水面中倒影的方法，在绘制倒影之前，首先需要关闭背面剪裁，同时调用pushMatrix方法保护现场，然后将3D物体移动至倒影的位置并绘制，最后恢复现场并打开背面剪裁。

20.7.4 赛艇类Boat

由于在本书前面章节中已经介绍了法向量类Normal、加载物体类LoadUtil与加载后的物体类LoadedObjectVertexNormal，因此这里不再赘述。故这里仅给出赛艇类的代码，具体内容如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的Boat.java。

```
package com.bn.clp;                                        //声明包名
......//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
public class Boat{
    List<LoadedObjectVertexNormal> lovo=new ArrayList<LoadedObjectVertexNormal>();
    public Boat(String [] str,MyGLSurfaceView mv,int programId){
        for(int i=0;i<str.length;i++){          //遍历赛艇部件数组，加载赛艇各个部件
            lovo.add(
                LoadUtil.loadFromFileVertexOnly(
                    str[i], mv.getResources(), programId
        ) ); } }
```

```
        public void drawSelf(float bx,float by,float bz,float yAngle,int dyFlag,int[] texId){
            for(int i=0;i<lovo.size();i++){                //遍历赛艇列表集合
                if(dyFlag==0){                             //绘制实体
                    MatrixState.pushMatrix();              //保护现场
                    MatrixState.translate(bx, by, bz);
                    MatrixState.rotate(yAngle, 0, 1, 0);
                    MatrixState.rotate(head_Angle, 1, 0, 0);
                    lovo.get(i).drawSelf(texId[i]);        //绘制赛艇各个部件
                    MatrixState.popMatrix();               //恢复现场
                }else if(dyFlag==1){                       //绘制倒影
                    float yTranslate=by;                   //实际绘制时 Y 的零点
                    float yjx=(WATER_HIGH_ADJUST-yTranslate)*2; //进行镜像绘制时的调整值
                    GLES20.glDisable(GLES20.GL_CULL_FACE);      //关闭背面剪裁
                    MatrixState.pushMatrix();              //保护现场
                    MatrixState.translate(bx, by, bz);
                    MatrixState.rotate(yAngle, 0, 1, 0);
                    MatrixState.rotate(-head_Angle, 1, 0, 0);
                    MatrixState.translate(0, yjx, 0);  //将赛艇沿 y 轴平移一定距离
                    MatrixState.scale(1, -1, 1);
                    lovo.get(i).drawSelf(texId[i]);        //绘制赛艇部件
                    MatrixState.popMatrix();               //恢复现场
                    GLES20.glEnable(GLES20.GL_CULL_FACE);//打开背面剪裁
    } } } }
```

• 第4行为本类成员变量的声明，主要是存储加载物体类的集合。

• 第5-10行为本类的构造器，主要是遍历存储赛艇部件的数组，通过LoadUtil调用loadFromFileVertexOnly方法加载赛艇的部件，并存储到List<LoadedObjectVertexNormal>集合。

• 第13-20行为绘制赛艇实体的代码，首先需要保护现场，然后将赛艇各个部件进行平移以及旋转并绘制，最后恢复现场。

• 第20-33行为绘制赛艇倒影的代码，首先需要关闭背面剪裁，同时调用pushMatrix方法保护现场，然后将赛艇各个部件移动至倒影的位置并绘制，最后恢复现场并打开背面剪裁。

20.7.5 可碰撞物体父类KZBJDrawer

前面小节中介绍了3D物体的父类BNDrawer的开发，接下来本小节将开发可碰撞物体父类——KZBJDrawer，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的KZBJDrawer.java。

```
package com.bn.clp;
public abstract class KZBJDrawer{        //可碰撞物件的父类，所有的可碰撞物体均是继承自该类
    public abstract void drawSelf(int texId);    //抽象的绘制类
}
```

> **说明**
>
> 可碰撞物体父类KZBJDrawer与3D物体的父类BNDrawer功能基本相同，都可以使代码具有非常高的可读性，且易于维护。

20.7.6 可碰撞物体控制类KZBJForControl

与前面介绍3D物体控制类TDObjectForControl类似，本小节也不再对可碰撞物体的生成代码开发进行介绍，本小节将直接介绍可碰撞物体控制类——KZBJForControl的开发，具体开发步骤如下。

（1）首先开发本类的整体框架，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的KZBJForControl.java。

```
package com.bn.clp;
......//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
public class KZBJForControl{          //可碰撞部件控制类,交通柱的 id 为 0，交通锥的 id 为 1
    KZBJDrawer kzbjdrawer;
    int id;                           //对应的碰撞物 id，0 表示交通筒，1 表示障碍物
    boolean state=false;              //false 表示可被碰撞，true 表示被撞后飞行中，否则不可碰撞
    float x; float y; float z;        //摆放的初始位置
    float alpha;                      //转动角度
    float alphaX; float alphaY; float alphaZ;        //转动轴向量
        float currentX; float currentY; float currentZ; //飞行中的当前位置
        int row; int col;                            //位置所在地图行与列
        float vx; float vy; float vz;                //飞行中的速度分量
        float time_Fly;                              //飞行累计时间
        MyActivity ma;                               //主控之类的引用
        public KZBJForControl(KZBJDrawer kzbjdrawer,int id,float x,float y,
                float z,int row,int col,MyActivity ma){
            this.kzbjdrawer=kzbjdrawer; this.id=id; //初始化本类的成员变量
            this.x=x; this.y=y; this.z=z;
            this.row=row; this.col=col; this.ma=ma;
        }
        public void drawSelf(int texId,int dyFlag){
            if(dyFlag==0){                           //绘制实体
                MatrixState.pushMatrix();
                if(!state){                          //原始状态绘制
                    MatrixState.translate(x, y, z);
                    ma.gameV.kzbj_array[id].drawSelf(texId);
                }else{                               //绘制飞行中物体
                    if(currentY>-40){                //如果已经飞行到地面以下，就不再绘制
                        MatrixState.translate(currentX, currentY, currentZ);
                        MatrixState.rotate(alpha,alphaX, alphaY, alphaZ);
                        ma.gameV.kzbj_array[id].drawSelf(texId);
                } }
                MatrixState.popMatrix();                     //恢复现场
            }else if(dyFlag==1){                             //绘制倒影
                float yTranslate=y;                          //实际绘制时 Y 的零点
                float yjx=(0-yTranslate)*2;                  //进行镜像绘制时的调整值
                MatrixState.pushMatrix();
                GLES20.glDisable(GLES20.GL_CULL_FACE);  //关闭背面剪裁
                if(!state){                                  //原始状态绘制
                    MatrixState.translate(x, y, z);     //将物体进行平移
```

```
41                  MatrixState.translate(0, yjx, 0);
42                  MatrixState.scale(1, -1, 1);
43                  ma.gameV.kzbj_array[id].drawSelf(texId);
44              }else{                              //飞行中绘制
45                  if(currentY>-40){               //如果已经飞行到地面以下，就不再绘制
46                      MatrixState.translate(currentX, currentY, currentZ);
47                      MatrixState.rotate(alpha,alphaX, alphaY, alphaZ);
48                      MatrixState.translate(0, yjx, 0);
49                      MatrixState.scale(1, -1, 1);
50                      ma.gameV.kzbj_array[id].drawSelf(texId);
51              } }
52              GLES20.glEnable(GLES20.GL_CULL_FACE);   //打开背面剪裁
53              MatrixState.popMatrix();                //恢复现场
54      } }
55      ……//此处省略了判断赛艇是否与可撞物体碰撞的方法，将在下面给出
56      public void go(){       //飞行移动方法，线程定时调用此方法，实现可撞物体飞行
57          if(!state){                         //如果不在飞行状态中不需要go
58              return;
59          }
60          time_Fly=time_Fly+0.3f;             //飞行持续时间增加
61          alpha=alpha+10;
62          currentX=x+vx*time_Fly; currentZ=z+vz*time_Fly;
63          currentY=y+vy*time_Fly-0.5f*5*time_Fly*time_Fly;//5为重力加速度
64          if(currentY<-2000){                 //当碰撞物体飞行落到地面以下2000时恢复原位
65              state=false;                    //state状态设置为false
66  } } }
```

- 第4-14行为本类成员变量的声明，主要包括碰撞物体的id、是否在飞行中的标志位、物体当前的位置、物体在飞行中的速度分量以及飞行的累计时间等。
- 第15-20行为本类的构造器，该构造器的主要作用是初始化本类中的部分成员变量。
- 第22-34行为绘制可碰撞物体的实体，由于可碰撞物体可能被赛艇碰撞从而飞起，因此需要分为静止中以及飞起两种状态绘制可碰撞物体。
- 第34-54行为绘制可碰撞物体在水面中的倒影，绘制倒影时也分为静止状态与飞行状态两种情况，其基本的绘制思想与绘制3D物体以及赛艇的倒影类似。
- 第56-66行为飞行中实时计算可碰撞物体位置的方法，首先判断物体是否在飞行中，如果不在飞行中则直接返回，否则根据时间实时计算可碰撞物体的位置。

（2）接下来开发本类中检测赛艇与碰撞物是否碰撞的方法，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的KZBJForControl.java。

```
1 public void checkColl(float bX,float bZ,float carAlphaTemp){
2     float bPointX=(float) (bX-BOAT_UNIT_SIZE*Math.sin(Math.toRadians(sight_angle)));
3     float bPointZ=(float) (bZ-BOAT_UNIT_SIZE*Math.cos(Math.toRadians(sight_angle)));
4     float carCol=(float) Math.floor((bPointX+UNIT_SIZE/2)/UNIT_SIZE);//计算碰撞点在的行与列
5     float carRow=(float) Math.floor((bPointZ+UNIT_SIZE/2)/UNIT_SIZE);
6     if(carRow==row&&carCol==col){  //如果大家在同一个格子里，进行严格的碰撞检测 KZBJBJ
7         double disP2=(bPointX-x)*(bPointX-x)+(bPointZ-z)*(bPointZ-z);
8         if(disP2<=4){                 //若船头距离目标小于 4 则为碰撞
9             if(SoundEffectFlag){
10                ma.shengyinBoFang(4, 0);
11            }
12            state=true;               //设置状态为飞行中状态
13            time_Fly=0;               //飞行持续时间清零
14            alpha=0; alphaX=(float) (-20*Math.cos(Math.toRadians(carAlphaTemp)));
15            alphaY=0;
16            alphaZ=(float) (20*Math.sin(Math.toRadians(carAlphaTemp)));
17            currentX=x;               //设置飞行起始点为原始摆放点
18            currentY=y; currentZ=z;
19            vx=(float) (-20*Math.sin(Math.toRadians(carAlphaTemp)));
20            vy=15; vz=(float) (-10*Math.cos(Math.toRadians(carAlphaTemp)));
21    } } }
```

- 第2-5行为计算赛艇碰撞点的 x 、z 坐标以及碰撞点在地图上的行与列。
- 第5-21行为检测赛艇是否与碰撞物发生碰撞。首先检测赛艇与碰撞物是否在同一个格子中，如果在同一个格子中则检测碰撞点与碰撞物中心点距离是否小于4,距离小于4的认为碰撞，就需要播放碰撞声音以及计算碰撞物的 x 反方向速度以及 y 方向速度。

20.7.7　赛艇与可碰撞物体检测碰撞线程ThreadColl

前一小节介绍了本案例中的可碰撞物体控制类的开发，接下来将开发本案例中赛艇与可碰撞物体检测碰撞线程——ThreadColl，具体代码如下。



代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的ThreadColl.java。

```java
1 package com.bn.clp;                                          //声明包名
2 import static com.bn.clp.MyGLSurfaceView.*;                  //导入相关类
3 public class ThreadColl extends Thread{
4     MyGLSurfaceView surface;                                 //游戏界面类的引用
5     public boolean flag=true;                                //线程中的循环标志位
6     public ThreadColl(MyGLSurfaceView surface){
7         this.surface=surface;                                //获取游戏界面类的对象
8         this.setName("ThreadColl");
9     }
10        public void run(){                                   //继承 Thread 重写的 run 方法
11            while(flag){
12                for(KZBJForControl kzbjfcTemp:surface.kzbjList){
13                    if(!kzbjfcTemp.state){      //判断可碰撞物体的碰撞标志位是否为 false
14                        kzbjfcTemp.checkColl(bxForSpecFrame,
15                                bzForSpecFrame, angleForSpecFrame);
16                    }
17                    kzbjfcTemp.go();                        //调用 go 方法使得物体运动
18                }
19                try{Thread.sleep(50);}catch(Exception e){e.printStackTrace();}
20    } } }
```

- 第4-5行为本类的成员变量的声明，主要包括游戏界面类的引用以及线程中的循环标志位。
- 第10-20行为继承Thread类重写的run方法，在该方法中要每隔50毫秒动态监测赛艇是否与可碰撞物体中的任何一个碰撞，并调用go方法计算撞飞物体的位置以及速度。

20.7.8　可吃物体父类SpeedForEat

前面小节中已经介绍了 3D物体的父类BNDrawer以及可碰撞物体的父类KZBJDrawer的代码开发，接下来在本小节中将开发可吃物体父类——SpeedForEat，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的SpeedForEat.java。

```java
package com.bn.clp;
public abstract class SpeedForEat{                     //所有可以吃掉物体的父类
    public abstract void drawSelf(int texId);          //抽象的绘制方法
}
```

说明

通过创建可吃物体的父类，可以非常容易地管理各个可吃的物体，同时易于添加其他可吃物体，增强代码的可读性，并且易于维

护。

20.7.9　可吃物体控制类SpeedForControl

由于本案例中的可吃物体的只有晶体一种，而生成晶体的代码非常简单，因此这里不再对本案例中的晶体类的开发进行介绍。这里只介绍与可吃物体有关的可吃物体的控制类——SpeedForControl的代码开发，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的SpeedForControl.java。

```java
package com.bn.clp;
......//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
public class SpeedForControl{                                   //可吃物体的控制类
    long pzTime;                                                //碰撞时的时间
    SpeedForEat speed;                                          //可吃物体的引用
    int id;                                                     //可吃物体id
    float x; float y; float z;                                  //可吃物体平移的距离
    static float angleY;                                        //可吃物体旋转的角度
    float rows; float cols;                                     //物体所在行与列
        boolean isDrawFlag=true;                                //是否绘制的标志位
        MyActivity ma;
        public SpeedForControl(SpeedForEat speedForEat,int id,float x,float y,float z,
                float angleY,float rows,float cols,MyActivity ma){
            this.speed=speedForEat; this.id=id;     //初始化本类的成员变量
            this.x=x; this.y=y; this.z=z;
            this.rows=rows; this.cols=cols;
            this.ma=ma;
        }
        public void drawSelf(int texId,int dyFlag){ //绘制方法
            if(isDrawFlag){
                if(dyFlag==0){                  //绘制实体
                    MatrixState.pushMatrix(); //保护现场
                    MatrixState.translate(x, y, z);
                    MatrixState.rotate(angleY, 0, 1, 0);
                    speed.drawSelf(texId);     //绘制可吃物体
                    MatrixState.popMatrix();   //恢复现场
```

```
27                }else if(dyFlag==1){                    //绘制倒影
28                    float yTranslate=y;                 //实际绘制时 Y 的零点
29                    float yjx=(0-yTranslate)*2;         //进行镜像绘制时的调整值
30                    MatrixState.pushMatrix(); //保护现场
31                    MatrixState.translate(x, y, z);
32                    MatrixState.rotate(angleY, 0, 1, 0);//将物体绕 y 轴旋转一定的角度
33                    MatrixState.translate(0, yjx, 0);  //将物体沿 y 轴平移一定的距离
34                    MatrixState.scale(1, -1, 1);
35                    speed.drawSelf(texId);              //绘制物体
36                    MatrixState.popMatrix();            //恢复现场
37        } } }
38        public void checkColl(float bX,float bZ,float carAlphaTemp){//计算是否发生碰撞
39            float bPointX=(float) (bX-BOAT_UNIT_SIZE*Math.sin(Math.toRadians(sight_
              angle)));
40            float bPointZ=(float) (bZ-BOAT_UNIT_SIZE*Math.cos(Math.toRadians(sight_
              angle)));
41            float carCol=(float) Math.floor((bPointX+UNIT_SIZE/2)/UNIT_SIZE);
42            float carRow=(float) Math.floor((bPointZ+UNIT_SIZE/2)/UNIT_SIZE);
43            if(carRow==rows&&carCol==cols&&isDrawFlag){
44                double disP2=(bPointX-x)*(bPointX-x)+(bPointZ-z)*(bPointZ-z);
45                if(disP2<=4){                           //碰撞了
46                    if(id==0){                          //吃了增加氮气的物件
47                        if(numberOfN2<maxNumberOfN2){
48                            numberOfN2=numberOfN2+1;  //氮气的数量值增加 1
49                        }
50                        if(SoundEffectFlag){        //打开声音
51                            ma.shengyinBoFang(3, 0);
52                        }
53                        isDrawFlag=false;           //绘制标志位设置为 false
54                        pzTime=System.currentTimeMillis();
55                    }else if(id==1){                    //吃掉减速
56                        if(SoundEffectFlag){        //打开声音
57                            ma.shengyinBoFang(3, 0);
58                        }
59                        CURR_BOAT_V=CURR_BOAT_V/2;//吃到减速物体速度减半
60                        isDrawFlag=false;           //绘制标志位设置为 false
61                        pzTime=System.currentTimeMillis();
62        } } } }
63        public void checkEatYet(){                      //如果经过 50 s, 则重新显示
64            if(!isDrawFlag&&((System.currentTimeMillis()-pzTime)%60000/1000>=50)){
65                isDrawFlag=true;                        //重新将绘制标志位设置为 true
66    } } }
```

- 第4-11行为本类中成员变量的声明，包括可吃物体的引用、可吃物体id、可吃物体平移的距离以及旋转的角度、可吃物体所在的行与列以及是否绘制的标志位。
- 第12-18行为本类的构造器，该构造器的主要作用是初始化本类的成员变量。
- 第19-37行为本类中的绘制方法，包括绘制实体以及绘制倒影，其中绘制倒影的方式与前面绘制3D物体倒影、绘制赛艇倒影的类似。
- 第38-62行为检测赛艇是否与可吃物体物发生碰撞。首先检测赛艇与碰撞物是否在同一个格 子中，如果在同一个格子中并且绘制标志位为true，则检测碰撞点与碰撞物中心点距离是否小于4，距离小于4的认

为碰撞，就需要播放碰撞声音，根据碰撞的种类增加氮气数量或者减小赛艇的速度，并设置绘制标志位为false。

- 第63-66行为判断被吃掉的物体是否已经经过50s，如果经过50s，则重新将绘制标志位设置为true。

20.7.10 晶体旋转线程RotateThread

前一小节介绍了可吃物体的控制类SpeedForControl类的开发，接下来本小节中将介绍晶体旋转线程——RotateThread的代码开发，该线程的主要作用是使晶体旋转，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的RotateThread.java。

```
1 package com.bn.clp;                                  //声明包名
2 ……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
3 public class RotateThread extends Thread{            //晶体旋转线程
4      public RotateThread(){                          //线程构造器
5           this.setName("RotateThread");
6      }
7      public void run(){                              //重写的run方法
8           while(Constant.threadFlag){
9                SpeedForControl.angleY=(SpeedForControl.angleY+6)%360;//计算晶体旋转角度
10                    if(CURR_BOAT_V==0){
11                         if(head_Angle<=head_Angle_Max&&KeyThread.upFlag){
12                              head_Angle=head_Angle+head_Angle_A;
13                              if(head_Angle==head_Angle_Max){
14                                   KeyThread.upFlag=false;
15                         } } else if(head_Angle>=-head_Angle_Max&&!KeyThread.upFlag){
16                              head_Angle=head_Angle-head_Angle_A;
17                              if(head_Angle==-head_Angle_Max){
18                                   KeyThread.upFlag=true;
19                    } } }else{
20                         if(head_Angle<4){
21                              head_Angle=head_Angle+head_Angle_A;
22                         }else{
23                              head_Angle=head_Angle-head_Angle_A;
24                    } }
25                    try{Thread.sleep(60);}catch(Exception e){e.printStackTrace();}
26      } } }
```

- 第9行为每隔一段时间定时更改晶体的旋转角度。
- 第10-19行为判断游戏中赛艇的速度是否为0，如果为0，则判断赛艇的上翘角度是否达到最大值，如果达到最大值，则设置upFlag为false，如果达到最小值，则设定upFlag为true。
- 第20-24行为判断赛艇的上翘角度是否小于4，如果小于4，则增加赛艇的上翘角度，否则减小赛艇的上翘角度。

20.7.11　赛艇与晶体碰撞检测线程ThreadForEat

前一小节介绍了本案例中的晶体旋转线程类的开发，接下来将开发赛艇与晶体碰撞检测线程—— ThreadForEat，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的ThreadForEat.java。

```
1 package com.bn.clp;                                   //声明包名
2 import static com.bn.clp.MyGLSurfaceView.*; //导入相关类
3 public class ThreadForEat extends Thread{    //检查与可吃掉的物体是否发生碰撞
4     MyGLSurfaceView surface;                          //游戏界面类的引用
5     public boolean flag=true;                          //线程中的循环标志位
6     public ThreadForEat(MyGLSurfaceView surface){
7          this.surface=surface;                         //获取游戏界面类的对象
8          this.setName("ThreadForEat");
9     }
10         @Override
11         public void run(){                            //继承 Thread 后重写的 run 方法
12              while(flag){
13                   for(SpeedForControl ksfcTemp:surface.speedWtList){
14                        if(ksfcTemp.isDrawFlag){    //判断此时的晶体是否被绘制
15                             ksfcTemp.checkColl(bxForSpecFrame, bzForSpecFrame, angleFor
                               SpecFrame);
16                        }else{                      //此时的晶体未绘制
17                             ksfcTemp.checkEatYet();
18                   } }
19                   try{Thread.sleep(50);}catch(Exception e){e.printStackTrace();}
20    } } }
```

- 第4-5行为本类的成员变量的声明，包括MyGLSurfaceView的引用以及线程中的循环标志位。
- 第13-18行为判断赛艇是否与可吃物体发生碰撞。主要是遍历存储可吃物体的集合，之后判断每一个可吃物体的绘制标志位是否为true。如果为true，则SpeedForControl调用checkColl方法判断赛艇与该可吃物体是否发生碰撞。不为true，则通过SpeedForControl调用checkEatYet方法判断是否应该将可吃物体的绘制标志位重新设置为true。

20.7.12　比较两灌木距离摄像机距离类MyComparable

前面小节介绍了3D物体、可碰撞物体以及可吃物体的开发，接下来将介绍本案例中所使用的树木的开发。本案例中使用的树木开发技术与前面章节中介绍到的树木开发技术不同。首先开发比较两树木距摄像机距离远近的类——MyComparable，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的MyComparable.java。

```
package com.bn.clp;                                    //声明包名
……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
public class MyComparable implements Comparator<float[]>{
    public int compare(float another[], float another2[]){
        float xs=another[0]-cxForSpecFrame;     //获取一个树木与摄像机在x轴上的偏移量
        float zs=another[1]-czForSpecFrame;     //获取一个树木与摄像机在z轴上的偏移量
        float xo=another2[0]-cxForSpecFrame;    //获取另一个树木与摄像机在x轴上的偏移量
        float zo=another2[1]-czForSpecFrame;    //获取另一个树木与摄像机在z轴上的偏移量
        float disA=(float)Math.sqrt(xs*xs+zs*zs);   //获取树木与摄像机的距离
        float disB=(float)Math.sqrt(xo*xo+zo*zo);
        return ((disA-disB)==0)?0:((disA-disB)>0)?-1:1; //判断两个树木与摄像机距离的大小
}}
```

说明

本案例中树木是通过多个纹理矩形拼接成的，因此需要检测组成树木的纹理矩形与摄像机的距离，再通过Collections调用sort方法对这些纹理矩形进行排序，并根据排序后的集合绘制树木，这样就不会发生遮挡现象。而本类正是用于比较组成树木的纹理矩形与摄像机的距离远近的比较器。

20.7.13　单个灌木类SingleShrub

本书前面章节中已经介绍了纹理矩形的开发，因此本小节不再介绍单个树木纹理矩形的开发，在本小节将直接介绍单个灌木类——SingleShrub的代码开发，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的SingleShrub.java。

```
1 package com.bn.clp;                                         //声明包名
2 ……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
3 public class SingleShrub{                                   //表示单个灌木的类
4     List<float []> all=new ArrayList<float []>();   //存储组成树木的6片纹理矩形的位置列表
5     ShrubForDraw sfd;                                       //用于绘制各个瓣的纹理矩形
6     float[] texList={                                       //纹理数组，每个树的纹理坐标可能不同
7         0,0, 0,1, 0.5f,0, 0.5f, 0,0,1, 0.5f,1
8     };
9     public SingleShrub(int programId){sfd=new ShrubForDraw(programId, texList);}
10        public void drawSelf(int texIds,int id,float x,float y,float z,int dyFlag){
11            float[][] tempData={                  //生成组成树木的6片纹理矩形的位置信息
12                {x,z,0}, {x+GRASS_UNIT_SIZE/2, z+GRASS_UNIT_SIZE*0.866f, 60},
13                {x+GRASS_UNIT_SIZE*1.5f, z+GRASS_UNIT_SIZE*0.866f, 120},
14                {x+GRASS_UNIT_SIZE*2, z, 180},
15                {x+GRASS_UNIT_SIZE*1.5f, z-GRASS_UNIT_SIZE*0.866f, 240},
16                {x+GRASS_UNIT_SIZE/2, z-GRASS_UNIT_SIZE*0.866f, 300}
17            };
18            all=Arrays.asList(tempData);
19            Collections.sort(all, new MyComparable());  //对6瓣进行排序
20            if(dyFlag==0){                               //绘制实体
21                for(float []tt:all){                     //循环列表进行树木的绘制
22                    MatrixState.pushMatrix();            //保护现场
23                    MatrixState.translate(tt[0], y, tt[1]);
24                    MatrixState.rotate(tt[2], 0, 1, 0);
25                    sfd.drawSelf(texIds);                //绘制纹理矩形
26                    MatrixState.popMatrix();             //恢复现场
27            } }else if(dyFlag==1){                       //绘制倒影
28                float yTranslate=y;                      //实际绘制时Y的零点
29                float yjx=(0-yTranslate)*2;              //进行镜像绘制时的调整值
30                for(float []tt:all){                     //循环列表进行树木的绘制
31                    MatrixState.pushMatrix();            //保护现场
32                    MatrixState.translate(tt[0], y, tt[1]);
33                    MatrixState.rotate(tt[2], 0, 1, 0);
```

```
34                    MatrixState.translate(0, yjx, 0);
35                    MatrixState.scale(1, -1, 1);
36                    sfd.drawSelf(texIds);                //绘制纹理矩形
37                    MatrixState.popMatrix();             //恢复现场
38    } } } }
```

- 第4-8行为本类中成员变量的声明，包括存储组成树木的6片纹理矩形的位置列表、单个树木纹理矩形的引用以及树木的纹理坐标数组。
- 第9行为本类的构造器，该构造器的主要作用是创建ShrubForDraw对象。
- 第11-19行为生成组成树木的6片纹理矩形的位置信息，将该信息添加到集合中，并对该集合进行排序。
- 第20-27行为绘制遍历存储单个树木纹理矩形的列表绘制树木，与前面小节介绍的绘制3D物体、可碰撞物体以及可吃物体相似，均需要首先保护现场，然后再绘制，最后恢复现场。
- 第27-38行为绘制树木在水面中的倒影。与前面小节介绍的绘制3D物体倒影、可碰撞物体倒影以及可吃物体倒影相似，均需要将物体移

动到倒影位置，然后再绘制。

20.7.14　灌木控制类ShrubForControl

前面小节介绍了单个灌木类SingleShrub的开发，接下来本小节将开发灌木控制类——ShrubForControl，该类的作用是比较两个灌木与摄像机的距离，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的ShrubForControl.java。

```
1  package com.bn.clp;                                           //声明包名
2  import static com.bn.clp.MyGLSurfaceView.*;                   //导入相关类
3  public class ShrubForControl implements Comparable<ShrubForControl>{
4      int rows; int cols;                                       //灌木所在的行与列
5      float xoffset; float yoffset; float zoffset;              //灌木的位置坐标
6      int id; int texIds;                                       //灌木编号以及纹理 id
7      ……//此处省略了本类的构造器，读者可以自行查阅随书光盘中的源代码
8      @Override
9      public int compareTo(ShrubForControl another){//重写的比较两个灌木离摄像机距离的方法
10             float xs=xoffset-cxForSpecFrame;          //计算灌木与摄像机在 x 轴上的偏移量
11             float zs=zoffset-czForSpecFrame;          //计算灌木与摄像机在 z 轴上的偏移量
12             float xo=another.xoffset-cxForSpecFrame;
13             float zo=another.zoffset-czForSpecFrame;
14             float disA=(float)Math.sqrt(xs*xs+zs*zs);   //计算灌木与摄像机的距离
15             float disB=(float)Math.sqrt(xo*xo+zo*zo);
16             return ((disA-disB)==0)?0:((disA-disB)>0)?-1:1; //比较两个灌木与摄像机的距离
17     } }
```

- 第4-6行为本类的成员变量的声明，包括灌木所在的行与列、灌木的位置坐标、灌木编号以及灌木所使用的纹理id。
- 第9-17行为实现Comparable接口后重写的compareTo方法，该方法的主要作用是比较两个灌木与摄像机的距离。

20.7.15　赛艇赛道类RaceTrack

前面开发了本案例中游戏界面使用到的部分3D物体、可碰撞物体以及可吃物体，接下来本小节将开发本案例中的赛艇赛道类——RaceTrack，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的RaceTrack.java。

```
package com.bn.clp;                                        //声明包名
……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
public class RaceTrack{                                    //代表赛艇赛道中竖直的赛道类
    private int mProgram;                                  //自定义渲染管线程序 id
    ……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
    boolean isZD;                                          //是否为直道标记
    public RaceTrack(int programId,float[][] yArray,int rows,int cols,boolean isZD){
        this.isZD=isZD;
        initVertexData(yArray,rows,cols,isZD);             //初始化顶点数据
            initShader(programId);                         //初始化 Shader
        }
        public void initVertexData(float[][] yArray,int rows,int cols,boolean isZD){
                                                           //初始化顶点数据的方法
            if(isZD){
                initVertexDataZD(yArray,rows,cols);//初始化直道顶点的数据信息
            }else{
                initVertexDataFZD(yArray,rows,cols);//初始化非直道顶点的数据信息
        } }
        ……//此处省略了初始化普通直道顶点数据的方法，读者可以自行查阅随书光盘中的源代码
        public void initVertexDataFZD(float[][] yArray,int rows,int cols){
                                                           //初始化非普通直道顶点数据的方法
            float width=UNIT_SIZE/cols; float height=UNIT_SIZE/rows;
                                                           //分成格子的宽度与高度
            vCount=rows*cols*2*3;                          //顶点的数量
            float[] vertex=new float[vCount*3];            //存储顶点坐标的数组
            float[] texture=new float[vCount*2];           //顶层顶点纹理坐标的数组
            float tempWidth=2.0f/cols;                     //分成格子的宽度
            float tempHeight=2.0f/rows;                    //分成格子的宽高度
            int countv=0; int countt=0;
            for(int i=0;i<rows;i++){                       //循环遍历顶点，行→列
                for(int j=0;j<cols;j++){
                    if(yArray[i][j]!=0||yArray[i+1][j]!=0||yArray[i][j+1]!=0){
                        vertex[countv++]=width*j-UNIT_SIZE/2;
                                                           //卷绕第一个三角形的左上角顶点
                        vertex[countv++]=yArray[i][j];
                        vertex[countv++]=height*i-UNIT_SIZE/2;
                        vertex[countv++]=width*j-UNIT_SIZE/2;
                                                           //卷绕第一个三角形的左下角顶点
                        vertex[countv++]=yArray[i+1][j];
                        vertex[countv++]=height*(i+1)-UNIT_SIZE/2;
                        vertex[countv++]=width*(j+1)-UNIT_SIZE/2;
                                                           //卷绕第一个三角形的右上角顶点
                        vertex[countv++]=yArray[i][j+1];
```

```
38                          vertex[countv++]=height*i-UNIT_SIZE/2;
39                          texture[countt++]=tempWidth*i-width/2;
                                                //获取第一个三角形的左上角纹理坐标
40                          texture[countt++]=tempHeight*j-height/2;
41                          texture[countt++]=tempWidth*(i+1)-width/2;
42                          texture[countt++]=tempHeight*j-height/2;
43                          texture[countt++]=tempWidth*i-width/2;
                                                //获取第一个三角形的右上角纹理坐标
44                          texture[countt++]=tempHeight*(j+1)-height/2;
45                      }
46                      if(yArray[i][j+1]!=0||yArray[i+1][j]!=0||yArray[i+1][j+1]!=0){
47                          vertex[countv++]=width*(j+1)-UNIT_SIZE/2;
                                                //卷绕第二个三角形的右上角顶点
48                          vertex[countv++]=yArray[i][j+1];
49                          vertex[countv++]=height*i-UNIT_SIZE/2;
50                          vertex[countv++]=width*j-UNIT_SIZE/2;
                                                //卷绕第二个三角形的左下角顶点
51                          vertex[countv++]=yArray[i+1][j];
52                          vertex[countv++]=height*(i+1)-UNIT_SIZE/2;
53                          vertex[countv++]=width*(j+1)-UNIT_SIZE/2;
                                                //卷绕第二个三角形的右下角顶点
54                          vertex[countv++]=yArray[i+1][j+1];
55                          vertex[countv++]=height*(i+1)-UNIT_SIZE/2;
56                          texture[countt++]=tempWidth*i-width/2;
                                                //获取第二个三角形的右上角纹理坐标
57                          texture[countt++]=tempHeight*(j+1)-height/2;
58                          texture[countt++]=tempWidth*(i+1)-width/2;
59                          texture[countt++]=tempHeight*j-height/2;
60                          texture[countt++]=tempWidth*(i+1)-width/2;
                                                //获取第二个三角形的右下角纹理坐标
61                          texture[countt++]=tempHeight*(j+1)-height/2;
62              } } }
63              ……//此处省略了本方法中的部分代码，读者可以自行查阅随书光盘中的源代码
64          }
65          ……//此处省略了初始化 Shader 的方法，读者可以自行查阅随书光盘中的源代码
66          ……//此处省略了绘制图形的方法的方法，读者可以自行查阅随书光盘中的源代码
67          ……//此处省略了真正的绘制任务的方法的方法，读者可以自行查阅随书光盘中的源代码
68      }
```

- 第7-11行为本类的构造器，该构造器的主要作用是初始化顶点纹理坐标以及初始化着色器程序。
- 第12-17行为本类中初始化顶点数据的方法，主要是调用initVertexDataZD方法初始化直道顶点的数据，调用initVertexDataFZD方法初始化非直道顶点的数据。
- 第19-64行为初始化非普通直道顶点数据的方法，主要是根据当前的行、列、单位格子的长度以及计算得出的赛艇赛道的高度数组，获得每个三角形的顶点坐标以及顶点纹理坐标，并将其存储到相应的缓冲区中，同时设置缓冲区起始位置。

说明

由于使用灰度图开发的赛艇赛道连接处不好处理，并且赛艇赛道有其独特的性质，因此本案例中采用代码生成的方式来开发赛艇赛道。

20.7.16　赛艇与桥碰撞检测类PZZ

前面小节中介绍了本案例中用到的3D物体的代码开发，其中包括桥，而本小节将于开发PZZ类就是检测赛艇与桥是否碰撞的类，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的PZZ.java。

```
1 package com.bn.clp;                                             //声明包名
2 public class PZZ{
3     final float UNIT_SIZE=2.1f;
4     float x; float z;                                           //赛艇的碰撞点坐标
5     float xL=UNIT_SIZE; float zL=2*UNIT_SIZE;                   //桥墩的 x 轴长度以及 z 轴长度
6     float row; float col;
7     float whichBridge;                                          //横向桥还是竖向桥的标志
8     public PZZ(float x,float z,float row,float col,float whichBridge){
9         this.x=x; this.z=z;                                     //获取赛艇的碰撞点坐标
10            this.row=row; this.col=col;                         //获取赛艇所在的行与列
11            this.whichBridge=whichBridge;                       //判断是横向桥还是竖向桥
12        }
13        public boolean isIn(float bx,float bz){                 //赛艇碰撞点是否在桥墩的范围内
14            if(whichBridge==0){
15                if(bx>=x&&bx<=x+xL&&bz>=z&&bz<=z+zL){
16                    return true;                                //赛艇碰撞点在桥墩内部返回 true
17                }else{
18                    return false;                               //赛艇碰撞点不再桥墩内部返回 false
19            } }else{
20                if(bx>=x&&bx<=x+zL&&bz>=z&&bz<=z+xL){
21                    return true;                                //赛艇碰撞点在桥墩内部返回 true
22                }else{
23                    return false;                               //赛艇碰撞点不再桥墩内部返回 false
24    } } } }
```

- 第3-7行为本类成员变量的声明，主要包括赛艇的碰撞点坐标、桥墩的 x 轴长度以及 z 轴长度、赛艇所在的行与列以及横向桥还是竖向桥的标志。
- 第8-12行为本类的构造器，该构造器的主要作用是初始化本类的成员变量。
- 第13-24行为检测赛艇碰撞点是否在桥墩的范围内的方法，首先需要判断此时的桥是横向还是竖向，然后判断赛艇碰撞点是否在桥墩内，如果在桥墩内则返回true，否则返回false。

20.7.17 赛艇运动路线工具类PathUtil

前一小节开发了赛艇与桥碰撞检测类PZZ，因此本小节将开发赛艇运动路线工具类——PathUtil，该类的主要作用是在赛艇自动巡航时，根据当前赛艇所在赛道的编号获取赛艇的运动路径，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的PathUtil.java。

```
1 package com.bn.clp;                                                //声明包名
2 ……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
3 public class PathUtil{
```

```
4    public static ArrayList<float[]> generatePath(){          //产生标准路径
5         ArrayList<float[]> path=new ArrayList<float[]>();     //存储路径的集合
6         for(int[] grid:PATH){                                //加入一个格子的路径
7              int type=MAP_ARRAY[grid[0]][grid[1]];           //获取此格子的类型编号
8              float[][] pathTemp=getSubPath(grid[0],grid[1],type);
9              for(float[] fa:pathTemp){
10                  path.add(fa);                              //在集合中添加路径信息
11             } }
12             return path;
13       }
14       public static float[][] getSubPath(int row,int col,int type){
                                                  //获取指定行列指定类型的赛道的路线
15             float[][] result=null;
16             float startX=col*UNIT_SIZE-UNIT_SIZE/2; //获取当前位置的起始 x 坐标
17             float startZ=row*UNIT_SIZE-UNIT_SIZE/2; //获取当前位置的起始 z 坐标
18             switch(type){
19                 case 0:                             //0 号赛道
20                     result=new float[2][2];
21                     result[0][0]=0; result[0][1]=UNIT_SIZE/2+3;
22                     result[1][0]=UNIT_SIZE; result[1][1]=UNIT_SIZE/2+3;
23                 break;
24                 ……//此处省略了其他类型赛道的路径，读者可以自行查阅随书光盘中的源代码
25             }
26             for(int i=0;i<result.length;i++){ //遍历路径，为其增加此时所在赛道的起始位置坐标
27                 result[i][0]=result[i][0]+startX;
28                 result[i][1]=result[i][1]+startZ;
29             }
30             return result;
31   } }
```

- 第4-13行为产生标准路径的方法，主要是遍历地图的矩阵路线，获取格子类型编号，调用getSubPath方法获取路径信息，同时遍历该标准路径添加到集合中。
- 第14-31行为获取指定行列指定类型的赛道路线的方法，该方法一开始首先需要获取当前赛道的起始*x* 坐标以及*z* 坐标，然后根据当前

赛道的编号获取该编号的标准路径，最后为获取的路径增加此时所在赛道的起始位置坐标即可。

> **说明**
>
> 由于本案例中赛艇的赛道分为16种，因此在考虑赛艇赛道情况时应该考虑16种。

20.7.18　尾浪类WeiLang

前一小节开发了赛艇运动路线工具类PathUtil，接下来本小节将开发尾浪类——WeiLang，该类主要作用是为赛艇添加尾浪，使得游戏更加真实，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的WeiLang.java。

```
package com.bn.clp;                                    //声明包名
……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
public class WeiLang{
    int mProgram;                                      //自定义渲染管线的引用
```

```
5     ……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
6     int vCount=0;                                              //顶点数量
7     public WeiLang(float a,float b,float height,float[] texCoor,int programId){
8         initVertexData(a,b,height,texCoor);                    //初始化顶点数据
9         initShader(programId);                                 //初始化着色器程序
10            new Thread(){                                      //启动一个线程定时换帧
11                public void run(){
12                    while(Constant.threadFlag){
13                        currStartST=(currStartST+0.1f)%1;//更改水面纹理起始点坐标
14                        try{Thread.sleep(100);}catch (InterruptedException e){e.
                          printStackTrace();}
15        } } }.start(); }
16        ……//此处省略了初始化顶点数据的方法，读者可以自行查阅随书光盘中的源代码
17        public void initShader(int programId){                 //初始化Shader的方法
18          mProgram = programId;                                //基于顶点着色器与片元着色器创建程序
19          maPositionHandle = GLES20.glGetAttribLocation(mProgram, "aPosition");
20          maTexCoorHandle= GLES20.glGetAttribLocation(mProgram, "aTexCoor");
21          muMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
22         maSTOffset=GLES20.glGetUniformLocation(mProgram, "stK");//尾浪纹理图偏移量的引用
23          muTMD=GLES20.glGetUniformLocation(mProgram, "tmd");//获取尾浪透明度的引用
24        }
25        ……//此处省略了本类中的绘制方法，读者可以自行查阅随书光盘中的源代码
26    }
```

- 第7-15行为本类的构造器，该构造器的主要作用是初始化顶点数据信息、初始化着色器程序以及开启线程定时更改尾浪纹理的起始纹理坐标。
- 第17-24行为初始化着色器程序的方法，其中需要获得顶点坐标的引用、顶点纹理坐标的引用、总变换矩阵的引用、尾浪纹理图偏移量的引用以及尾浪透明度的引用。

20.7.19　水面类Water

前面小节中开发了本案例中的尾浪类WeiLang，接下来本小节将开发水面类Water，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的Water.java。

```
1 package com.bn.clp;                                    //声明包名
2 ......//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
3 public class Water{                                     //有流动效果的水面
4     int mPrograms;                                      //自定义渲染管线程序引用
5     ......//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
6     float currStartST=0;                                //水面纹理坐标的当前起始坐标 0~1
7     public Water(int programId,int rows,int cols){
8         initVertexData(rows,cols);                      //初始化顶点坐标与着色数据
9         initShader(programId);                          //初始化 shader
10            new Thread(){                               //启动一个线程定时换帧
11                public void run(){
12                    while(Constant.threadFlag){
13                        currStartST=(currStartST+0.004f)%1;
```

```
14                        try{Thread.sleep(100);}catch (InterruptedException e){e.
                          printStackTrace();}
15        } } }.start(); }
16    ......//此处省略了初始化顶点数据的方法，读者可以自行查阅随书光盘中的源代码
17    public void initShader(int programId){              //初始化 shader
18        mPrograms =programId;                           //基于顶点着色器与片元着色器创建程序
19        maPositionHandle = GLES20.glGetAttribLocation(mPrograms, "aPosition");
20        maTexCoorHandle= GLES20.glGetAttribLocation(mPrograms, "aTexCoor");
21        muMVPMatrixHandle = GLES20.glGetUniformLocation(mPrograms, "uMVPMatrix");
22        maSTOffset=GLES20.glGetUniformLocation(mPrograms, "stK");
23    }
24    ......//此处省略了本类中的绘制方法，读者可以自行查阅随书光盘中的源代码
25    public float[] generateTexCoor(int bw,int bh){     //自动切分纹理产生纹理数组的方法
26        float[] result=new float[bw*bh*6*2];
27        float sizew=1.0f/bw; float sizeh=1.0f/bh;      //列数与行数
28        int c=0;
29        for(int i=0;i<bh;i++){//每行列一个矩形，由两个三角形构成，共 6 个点，12 个纹理坐标
30            for(int j=0;j<bw;j++){
31                float s=j*sizew; float t=i*sizeh;
32                result[c++]=s; result[c++]=t;               //获取顶点纹理坐标
33                result[c++]=s; result[c++]=t+sizeh;         //获取顶点纹理坐标
34                result[c++]=s+sizew; result[c++]=t;         //获取顶点纹理坐标
35                result[c++]=s+sizew; result[c++]=t;         //获取顶点纹理坐标
36                result[c++]=s; result[c++]=t+sizeh;         //获取顶点纹理坐标
37                result[c++]=s+sizew; result[c++]=t+sizeh;
38        } }
39        return result;
40  } }
```

- 第7-15行为本类的构造器，该构造器的主要作用是初始化顶点数据信息、初始化着色器程序以及开启线程定时更改水面纹理的起始纹理坐标。
- 第17-23行为初始化着色器程序的方法，其中需要获得顶点坐标的引用、顶点纹理坐标的引用、总变换矩阵的引用以及水面纹理图偏移量的引用。
- 第25-40行为自动切分纹理产生纹理数组的方法，该方法开始首先需要创建存储纹理坐标的数组，然后根据切分的行与列计算相应的顶点纹理坐标。

20.7.20　游戏界面类MyGLSurfaceView

本案例中条幅生成使用的是纹理矩形与圆柱拼成的，天空穹的绘制与前面章节中介绍到的球体的绘制相同，加速按钮类、绘制时间、计圈、氮气、暂停、换人称视角以及加速按钮等均是纹理矩形，只是纹理坐标不相同而已，所以这里就不再对其进行介绍。接下来本小节将开发游戏界面，具体步骤如下。

（1）介绍本案例的具体开发代码之前，首先开发本类的主要框架，其代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的MyGLSurfaceView.java。

```
package com.bn.clp;                                  //声明包名
......//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
```

```
public class MyGLSurfaceView extends GLSurfaceView{
    public static long betweenStartAndPauseTime;      //游戏中记录暂停时已经用的时间
    ……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
    public boolean isShaChe=false;                    //是否在刹车的标志位
    ……//此处省略了本类中加载 Bitmap 位图资源的方法，读者可以自行查阅随书光盘中的源代码
    ……//此处省略了触摸监听器的方法，将在下面介绍
    public MyGLSurfaceView(Context context){
            super(context);
            ma=(MyActivity)context;                   //获取主控制类的引用
            this.setKeepScreenOn(true);               //保持屏幕常量
            if(isSpeedMode){                          //判断是否为竞速模式
                otherBoatLocation[0][0]=YACHT_INI_X-3;//从常量类中加载其他船的初始位置
                otherBoatLocation[0][1]=YACHT_INI_Z-3;
                otherBoatLocation[0][2]=0;
                otherBoatLocation[1][0]=YACHT_INI_X+3;
                otherBoatLocation[1][1]=YACHT_INI_Z-4;
                otherBoatLocation[1][2]=0;
            }
            bx=YACHT_INI_X; bz=YACHT_INI_Z;           //从常量类中加载英雄船的初始位置
            cx=(float)(bx+Math.sin(Math.toRadians(sight_angle))*DISTANCE);;
                                                      //摄像机 x 坐标
            cy=CAMERA_INI_Y;                          //摄像机 y 坐标
            cz=(float)(bz+Math.cos(Math.toRadians(sight_angle))*DISTANCE);
                                                      //摄像机 z 坐标
            tx=(float)(cx-Math.sin(Math.toRadians(sight_angle))*DISTANCE);
                                                      //观察目标点 x 坐标
            ty=CAMERA_INI_Y-1.5f;                     //平视观察目标点 y 坐标
            tz=(float)(cz-Math.cos(Math.toRadians(sight_angle))*DISTANCE);
                                                      //观察目标点 z 坐标
            keyState=keyState|0x1;                    //使小船永远向前走
            this.setEGLContextClientVersion(2);       //设置使用 ES2.0
            sRenderer=new SceneRenderer();
            setRenderer(sRenderer);                   //设置渲染器
            setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
            kt=new KeyThread(this,ma);                //创建模拟的键盘监控线程
            tc=new ThreadColl(this);                  //创建检测赛艇是否与可撞物体碰撞的线程
            tfe=new ThreadForEat(this);               //创建检测赛艇是否与可吃物体碰撞的线程
        }
        private class SceneRenderer implements GLSurfaceView.Renderer{ //创建渲染器
            BackGround bgd;                           //新添加的
            ……//此处省略了本渲染内部类中的部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
            float stK;
            ……//此处省略了本渲染类中的绘制方法，读者可以自行查阅随书光盘中的源代码
            ……//此处省略了绘制后视镜的方法，将在下面介绍
            ……//此处省略了绘制后视镜框的方法，读者可以自行查阅随书光盘中的源代码
            ……//此处省略了绘制小雷达的方法的方法，将在下面介绍
            ……//此处省略了绘制倒计时的方法的方法，读者可以自行查阅随书光盘中的源代码
            ……//此处省略了绘制开始以及结束标志的方法的方法，读者可以自行查阅随书光盘中的源代码
            ……//此处省略了绘制计时器、lap 以及仪表板的方法的方法，将在下面介绍
```

```
48              ……//此处省略了绘制各个编号赛道的方法，读者可以自行查阅随书光盘中的源代码
49              ……//此处省略了绘制赛道中的水面的方法，读者可以自行查阅随书光盘中的源代码
50              ……//省略了绘制赛道预备、根据地图矩阵绘制赛道以及根据地图矩阵绘制赛道中的水面的方法
51              ……//此处省略了重写的 onSurfaceChanged 方法与 onSurfaceCreated 方法，将在下面介绍
52              ……//此处省略了加载数据的方法，读者可以自行查阅随书光盘中的源代码
53          }
54          ……//此处省略了根据图片 id 获得纹理的 id 的方法，读者可以自行查阅随书光盘中的源代码
55          ……//此处省略了根据 Bitamp 位图获得纹理的 id 的方法，读者可以自行查阅随书光盘中的源代码
56          ……//此处省略了绘制 3D 部件以及树木部件的相关方法，将在下面介绍
57          ……//此处省略了可碰撞物体(交通锥、交通柱)部件预备的方法，读者可以自行查阅随书光盘中的源代码
58          ……//此处省略了绘制可碰撞物体的方法的方法，读者可以自行查阅随书光盘中的源代码
59          ……//此处省略了初始化 3D 部件集合的方法，读者可以自行查阅随书光盘中的源代码
60          ……//此处省略了初始化树木的方法，读者可以自行查阅随书光盘中的源代码
61          ……//此处省略了初始化可碰撞物件列表集合的方法，读者可以自行查阅随书光盘中的源代码
62          public static long gameContinueTime(){       //获取游戏时间
63              return System.currentTimeMillis()-gameStartTime;
64          }
65          ……//此处省略了初始化可以吃掉物体的列表的方法，读者可以自行查阅随书光盘中的源代码
66          ……//此处省略了加减速预备的方法，读者可以自行查阅随书光盘中的源代码
67          ……//此处省略了绘制加速、减速箱子的方法，读者可以自行查阅随书光盘中的源代码
68      }
```

- 第13-20行为判断是否为竞速模式，如果为竞速模式，则从常量类中加载其他船的初始位置。
- 第21-27行为从常量类中加载英雄船的初始位置，计算摄像机的位置坐标以及摄像机目标点的位置坐标。
- 第29-30行为指明使用OpenGL ES 2.0，创建渲染器，设置渲染器，并设置渲染模式为主动渲染。
- 第33-35为创建模拟的键盘监控线程、创建检测赛艇是否与可撞物体碰撞的线程以及创建赛艇是否与可吃物体碰撞的线程。
- 第62-64行为获取游戏时间的方法，主要是通过System调用currentTimeMillis获取当前时间，并减去游戏开始的时间。

（2）接下来开发本类中的触控监听方法，主要是加速按钮、刹车按钮、暂停按钮以及是否打开后视镜的区域，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的MyGLSurfaceView.java。

```
@Override
public boolean onTouchEvent(MotionEvent e){        //重写的触控方法
    float x=e.getX();                              //获取触控点 x 坐标
    float y=e.getY();                              //获取触控点 y 坐标
    float yRatio=y/com.bn.st.xc.Constant.SCREEN_HEIGHT;
    float xRatio=x/com.bn.st.xc.Constant.SCREEN_WIDTH;
    switch(e.getAction()){
        case MotionEvent.ACTION_DOWN:              //触控事件为按下
            if(isAllowToClick){
                    if(xRatio>Self_Adapter_Data_ON_TOUCH[screenId][10]&&
                        xRatio<Self_Adapter_Data_ON_TOUCH[screenId][11]
```

```
&&yRatio>Self_Adapter_Data_ON_TOUCH[screenId][8]&&
yRatio<Self_Adapter_Data_ON_TOUCH[screenId][9]&&!isPaused){
                                    //加速按钮
    if(numberOfN2>0){       //判断氮气的数量是否大于 0
        if(SoundEffectFlag){
            ma.shengyinBoFang(2, 0);//开启声音
        }
        numberOfN2=numberOfN2-1;  //氮气数量减 1
        Max_BOAT_V=Max_BOAT_V_FINAL;
        kt.dqCount=80;
    }else{
        numberOfN2=0;
} }else if(xRatio>Self_Adapter_Data_ON_TOUCH[screenId][6]&&
        xRatio<Self_Adapter_Data_ON_TOUCH[screenId][7]
        &&yRatio>Self_Adapter_Data_ON_TOUCH[screenId][4]&&
        yRatio<Self_Adapter_Data_ON_TOUCH[screenId][5]){
    if(isPaused){                //暂停态到运行态
        CURR_BOAT_V=CURR_BOAT_V_PAUSE;
        BOAT_A=BOAT_A_PAUSE;//船的加速度
        DEGREE_SPAN=2f;
        CURR_BOAT_V_PAUSE=0;
        BOAT_A_PAUSE=0;
        isPaused=false;
        kt.moveFlag=true;
        KeyThread.otherBoatFlag=true;
        gameStartTime=System.currentTimeMillis();
    }else if(!isPaused){             //运行态到暂停态
        CURR_BOAT_V_PAUSE=CURR_BOAT_V;
        BOAT_A_PAUSE=BOAT_A;
        CURR_BOAT_V=0;
        BOAT_A=0;
        DEGREE_SPAN=0;
        isPaused=true;
        kt.moveFlag=false;
        KeyThread.otherBoatFlag=false;
        betweenStartAndPauseTime=gameContinueTime()+
            betweenStartAndPauseTime;
        CURR_BOAT_V_TMD=0;
} }else if(xRatio>Self_Adapter_Data_ON_TOUCH[screenId][14]&&
        xRatio<Self_Adapter_Data_ON_TOUCH[screenId][15]
        &&yRatio>Self_Adapter_Data_ON_TOUCH[screenId][12]&&
        yRatio<Self_Adapter_Data_ON_TOUCH[screenId][13]&&!
        isPaused){
    isShaChe=true;               //按下刹车按钮，设置标志位为 true
    BOAT_A=-0.02f;               //加速度为-0.02f
}else if(xRatio>Self_Adapter_Data_ON_TOUCH[screenId][18]&&
        xRatio<Self_Adapter_Data_ON_TOUCH[screenId][19]
        &&yRatio>Self_Adapter_Data_ON_TOUCH[screenId][16]&&
        yRatio<Self_Adapter_Data_ON_TOUCH[screenId][17]){
    if(isOpenHSJ){               //当前打开后视镜标志位为 true
        isOpenHSJ=false;
    }else{                       //当前打开后视镜标志位为 false
        isOpenHSJ=true;
```

```
63                 } } }
64             break;
65             case MotionEvent.ACTION_UP:              //触控事件为抬起
66                 if(isShaChe&&!isPaused){
67                     isShaChe=false;                  //设置刹车标志位为 false
68                     BOAT_A=0.025f;                   //设置加速度值
69                 }
70             break;
71         }
72         return true;
73     }
```

- 第3-6行为本方法中的临时变量，主要是获取触控点的坐标，同时该触控点所占屏幕的比例。
- 第9-23行为判断单击的是否为加速按钮，如果为加速按钮，则判断氮气的数量是否大于0，如果大于零，则开启声音、氮气数量减1并且使赛艇加速。
- 第23-49行为判断单击的是否为暂停按钮，并且根据本游戏的当前状态更改赛艇当前的速度以及加速度等值。
- 第49-55行为判断单击的是否为刹车按钮，如果为刹车按钮，则设置刹车标志位isShaChe为true，并且设置加速度为-0.02f。
- 第55-63行为判断单击的是否为后视镜区域，如果为后视镜区域，则判断打开后视镜标志位isOpenHSJ的值，并将其置反。
- 第65-70行为检测屏幕触控事件为抬起，则判断是否单击了刹车按钮并且游戏正在运行，则设置刹车标志位为false，同时设置赛艇加速度的值。

（3）接下来将开发本游戏中绘制倒视镜中内容的方法，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的MyGLSurfaceView.java。

```
public void drawHouShiJing(){
    GLES20.glEnable(GL10.GL_SCISSOR_TEST);                        //启用剪裁测试
    GLES20.glScissor((int)Self_Adapter_Data_HSJ_XY[screenId][0],
            (int)Self_Adapter_Data_HSJ_XY[screenId][1],300,90); //设置显示区域
   GLES20.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
   MatrixState.setProjectFrustum(-1.0f*ratio, ratio, -1.7f*0.7f, 0.3f*0.7f, 1, 40(
    MatrixState.setCamera(                  //调用此方法产生摄像机 9 参数位置矩阵
        cx, cy, cz,
        (float)(cx+Math.sin(Math.toRadians(sight_angle))*DISTANCE),
            ty+1.1f,
            (float)(cz+Math.cos(Math.toRadians(sight_angle))*DISTANCE),
            0, 1, 0
        );
        MatrixState.copyMVMatrix();                        //复制矩阵
        sky.drawSelf(sky_texId,bxForSpecFrame, bzForSpecFrame,1);
        sky.drawSelf(sky_texId,bxForSpecFrame, bzForSpecFrame,0);
        GLES20.glDisable(GLES20.GL_CULL_FACE);             //关闭背面剪裁
        drawSD(startST);                                   //绘制赛道
        GLES20.glEnable(GLES20.GL_CULL_FACE);              //打开背面剪裁
        DrawTDObjects(1);                                  //绘制 3D 物体的倒影
        DrawTDObjects(0);                                  //绘制 3D 物体
```

```
boat.drawSelf(bxForSpecFrame, 0.3f, bzForSpecFrame,angleForSpecFrame,1,
heroBoatTexId);
if(isSpeedMode){
  for(int i=0;i<otherBoatLocationForHelp.length;i++){//绘制其他船
    if(i==0){
        quickBoat.drawSelf(otherBoatLocationForHelp[i][0], 0.3f,
            otherBoatLocationForHelp[i][1],otherBoatLocationForHelp[i]
            [2],1,quickBoatTexId);
    }else if(i==1){
        slowBoat.drawSelf(otherBoatLocationForHelp[i][0], 0.3f,
                otherBoatLocationForHelp[i][1],otherBoatLocationForHelp[i]
                [2],1,slowBoatTexId);
} } }
drawKZBJ(1);                                    //绘制可碰撞的交通柱与交通锥的倒影
GLES20.glDisable(GLES20.GL_CULL_FACE);          //关闭背面剪裁
GLES20.glEnable(GLES20.GL_BLEND);               //开启混合
GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
drawTrees(1);                                   //绘制树倒影
GLES20.glDisable(GLES20.GL_BLEND);              //关闭混合
GLES20.glEnable(GLES20.GL_BLEND);               //开启混合
GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
drawSpeedForEat(1);                             //绘制可吃物体倒影
GLES20.glDisable(GLES20.GL_BLEND);              //关闭混合
drawStartAndEnd(numberOfTurns,1);               //绘制开始以及结束标志倒影
GLES20.glEnable(GLES20.GL_BLEND);               //开启混合
GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
drawWater(startST);                             //绘制水面
GLES20.glDisable(GLES20.GL_BLEND);              //关闭混合
GLES20.glEnable(GLES20.GL_CULL_FACE);           //打开背面剪裁
drawKZBJ(0);                                    //绘制可碰撞的交通柱与交通锥
boat.drawSelf(bxForSpecFrame, 0.3f, bzForSpecFrame,angleForSpecFrame,0,
heroBoatTexId);
if(isSpeedMode){
  for(int i=0;i<otherBoatLocationForHelp.length;i++){    //绘制其他船
    if(i==0){
        quickBoat.drawSelf(otherBoatLocationForHelp[i][0], 0.3f,
            otherBoatLocationForHelp[i][1],otherBoatLocationForHelp[i]
            [2],0,quickBoatTexId);
    }else if(i==1){
        slowBoat.drawSelf(otherBoatLocationForHelp[i][0], 0.3f,
            otherBoatLocationForHelp[i][1],otherBoatLocationForHelp[i]
            [2],0,slowBoatTexId);
} } }
MatrixState.pushMatrix();
GLES20.glEnable(GLES20.GL_BLEND);               //开启混合
GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
MatrixState.translate(bxForSpecFrame, 0.02f, bzForSpecFrame);
MatrixState.rotate(-90, 1, 0, 0);
MatrixState.rotate(angleForSpecFrameS, 0, 0, 1);
wl.drawSelf(waterId,stK);                       //绘制尾浪
GLES20.glDisable(GLES20.GL_BLEND);              //关闭混合
MatrixState.popMatrix();
```

```
68        GLES20.glDisable(GLES20.GL_CULL_FACE); //关闭背面剪裁
69        GLES20.glEnable(GLES20.GL_BLEND);      //开启混合
70        GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
71        drawTrees(0);                          //绘制树
72        GLES20.glDisable(GLES20.GL_BLEND);     //关闭混合
73        drawStartAndEnd(numberOfTurns,0);
74        GLES20.glEnable(GLES20.GL_BLEND);      //开启混合
75        GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
76        drawSpeedForEat(0);
77        GLES20.glDisable(GLES20.GL_BLEND);     //关闭混合
78        GLES20.glDisable(GL10.GL_SCISSOR_TEST);//禁用剪裁测试
79    }
```

- 第2-4行为实现倒视镜功能启用了剪裁测试并设置了剪裁测试的允许区域。
- 第5-13行为清除颜色缓存与深度缓存、设置投影模式为透视投影以及设置摄像机的位置。
- 第14-77行为绘制游戏中的各个部件及其倒影，而竞速模式与计时模式不同，需要多绘制两艘赛艇因此该方法中需要判断是否为竞速模式。
- 第78行为禁用剪裁测试，开发过程中如果打开了剪裁测试，那么在最后必须禁用剪裁测试。

（4）接下来将开发绘制小雷达的方法。在竞速模式中右上角可以看到开发的小雷达，通过其可以看出自己使用的赛艇与其他赛艇之间相差的距离。具体开发代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的MyGLSurfaceView.java。

```
1 public void drawRadar(){                                      //绘制小雷达的方法
2     MatrixState.pushMatrix();                                 //保护现场
3    GLES20.glEnable(GLES20.GL_BLEND);                          //开启混合
4    GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
5    MatrixState.translate(Self_Adapter_Data_TRASLATE[screenId][12],
6          Self_Adapter_Data_TRASLATE[screenId][13], -2);
7     radar_Background.drawSelf(radarBackGroundTexId);
8     MatrixState.popMatrix();                                  //恢复现场
9     MatrixState.pushMatrix();                                 //保护现场
10        MatrixState.translate(Self_Adapter_Data_TRASLATE[screenId][12],
11                Self_Adapter_Data_TRASLATE[screenId][13], -1);
12        MatrixState.rotate(sight_angle, 0, 0, 1);
13        radar_Zhizhen.drawSelf(radarZhiZhenTexId);   //绘制表示自身赛艇的微型赛艇
14        MatrixState.popMatrix();                              //恢复现场
15        for(int i=0;i<otherBoatLocation.length;i++){
16            float x_Temp=otherBoatLocationForHelp[i][0]-bxForSpecFrame;
17            float z_Temp=otherBoatLocationForHelp[i][1]-bzForSpecFrame;
18            float r_Temp=(float) Math.sqrt((x_Temp/Radar_Ratio)*(x_Temp/Radar_Ratio)+
19                         (z_Temp/Radar_Ratio)*(z_Temp/Radar_Ratio));
20            if(r_Temp<=0.26f){
21                MatrixState.pushMatrix();                     //保护现场
22                MatrixState.translate(Self_Adapter_Data_TRASLATE[screenId][12],
23                        Self_Adapter_Data_TRASLATE[screenId][13], -1f);
24                MatrixState.translate(x_Temp/Radar_Ratio, -z_Temp/Radar_Ratio, 0);
25                other_Radar_Zhizhen.drawSelf(radarOtherBoatTexId);
26                MatrixState.popMatrix();                      //恢复现场
27    }}}
```

- 第2-8行为绘制该小雷达的背景。
- 第9-14行为绘制小雷达上面表示绘制自己所用赛艇的微型赛艇。
- 第15-27行为遍历其他赛艇位置获得与自己所用赛艇的距离，然后绘制表示其他赛艇的圆点。

（5）接下来将开发计算游戏用时、绘制游戏圈数的方法以及绘制仪表板的方法，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的MyGLSurfaceView.java。

```
1 public void drawTimeAndLap(){                          //计算游戏用时与绘制游戏圈数
2     if(!isJiShi){
3         dt.toTotalTime(0);                             //计算游戏进行时间
4     }else if(isPaused){                                //暂停状态
5         dt.toTotalTime(betweenStartAndPauseTime);
6     }else if(!isPaused){                               //非暂停态，运行时
7         if(numberOfTurns<3){
8             dt.toTotalTime(MyGLSurfaceView.gameContinueTime()+betweenStartAndPause
              Time);
9             gameTimeUse=MyGLSurfaceView.gameContinueTime()+betweenStartAndPauseTime;
10            }else{
11                dt.toTotalTime(gameTimeUse);
12        } }
13        MatrixState.pushMatrix();                      //保护现场
14        GLES20.glEnable(GLES20.GL_BLEND);              //开启混合
15        GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
16        MatrixState.translate(-0.35f, 0.8f, -2);       //设置平移矩阵
17        if(numberOfTurns<=2){
18            dt.drawSelf(timeTexId,numberOfTurns,numberOfN2,goTexId,shacheTexId,
              isShaChe);
19        }else{
20            dt.drawSelf(timeTexId,2,numberOfN2,goTexId,shacheTexId,isShaChe);
21        }
22        GLES20.glDisable(GLES20.GL_BLEND);             //关闭混合
23        MatrixState.popMatrix();                       //恢复现场
24    }
25    public void drawYiBiaoBan(){                       //绘制仪表板的方法
26        db.changeangle(CURR_BOAT_V);                   //根据当前速度，得到指针旋转的角度
27        MatrixState.setProjectOrtho(-ratio, ratio, -1, 1, 1, 100);
28        MatrixState.setCamera(0, 0, 0, 0, 0, -1, 0, 1, 0);  //设置摄像机基本位置
29        MatrixState.copyMVMatrix();                    //复制摄像机矩阵
30        MatrixState.pushMatrix();
31        GLES20.glEnable(GLES20.GL_BLEND);              //开启混合
32        GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
33        MatrixState.translate(Self_Adapter_Data_TRASLATE[screenId][9],
34            Self_Adapter_Data_TRASLATE[screenId][10], -2);
35        db.drawSelf(ybbTexId);                         //绘制仪表板
36        GLES20.glDisable(GLES20.GL_BLEND);             //关闭混合
37        MatrixState.popMatrix();
38    }
```

- 第1-24行为计算游戏用时与绘制游戏圈数的方法，其中在计算游戏耗费时间时需要判断是否为暂停状态，暂停状态时不需要计算游戏用时。在绘制游戏圈数时需要打开混合并设置混合因子。
- 第25-38行为绘制游戏中使用到的仪表板的方法，该方法开始需要根据当前的速度获得指针旋转的角度，在绘制仪表板时也需要打开混合并设置混合因子。

（6）接下来将开发准备赛道数据、根据地图矩阵绘制赛道以及根据地图矩阵绘制赛道水面的方法，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的MyGLSurfaceView.java。

```
1 public void sdYb(){                                              //赛道预备数据
2     sdCount=0;                                                   //计数器
3     int mrow=MAP_ARRAY.length; int mcol=MAP_ARRAY[0].length;
4     for(int i=0;i<mrow;i++){                                     //循环行
5         int rowM=i-bRow;
6         if(rowM>=NUMBER_MAP||rowM<=-NUMBER_MAP){
7             continue;                                            //不在规定的格子内，执行下一个
8         }
9         for(int j=0;j<mcol;j++){                                 //循环列
10                int colM=j-bCol;
11                if(colM>=NUMBER_MAP||colM<=-NUMBER_MAP){
12                    continue;                                    //不在规定的格子内，执行下一个
13                }
14                if(ClipGrid.CLIP_MASK[directNo][rowM+2][colM+2]){
15                    sdHz[sdCount][0]=i;                          //记录行号
16                    sdHz[sdCount][1]=j;                          //记录列号
17                    sdCount++;                                   //计数器值加1
18    } } } }
19    public void drawSD(float startST){                           //根据地图矩阵绘制赛道
20        for(int i=0;i<sdCount;i++){
21            drawSDSingle(
22                MAP_ARRAY[sdHz[i][0]][sdHz[i][1]],               //赛道编号
23                sdHz[i][0], sdHz[i][1]                           //赛道块在地图上的行与列
24    ); } }
25    public void drawWater(float startST){                        //根据地图矩阵绘制赛道中的水面
26        for(int i=0;i<sdCount;i++){
27            if(MAP_ARRAY[sdHz[i][0]][sdHz[i][1]]!=8){
28                drawWaterSingle(
29                    sdHz[i][0], sdHz[i][1],                      //赛道块在地图上的行与列
30                    startST
31    ); } } }
```

- 第1-18行为准备赛道数据的方法，由于游戏中的地图矩阵可能非常大，因此不可能将游戏中地图与地图上的物体在每一时刻都全部绘制，这时就仅需要绘制玩家赛艇所在部分地图与地图上的物体。
- 第19-24行为根据通过调用sdYb准备的赛道数据绘制游戏中赛道的方法。
- 第25-31行为根据通过调用sdYb准备的赛道数据绘制游戏中赛道中水面的方法。

（7）接下来开发实现GLSurfaceView.Renderer接口后重写的onSurfaceChanged方法以及onSurfaceCreated方法，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的MyGLSurfaceView.java。

```java
@Override
public void onSurfaceChanged(GL10 gl, int width, int height){
    GLES20.glViewport(0, 0, width, height);                  //设置视窗大小及位置
    ratio=(float)width/height;                               //计算 GLSurfaceView 的宽高比
}
@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config){
    synchronized(MyActivity.boatInitLock){
        GLES20.glClearColor(0.6039f,0.9333f,0.9843f,1.0f);  //设置屏幕背景色 RGBA
            GLES20.glEnable(GLES20.GL_CULL_FACE);           //打开背面剪裁
            GLES20.glEnable(GLES20.GL_DEPTH_TEST);          //打开深度检测
            MatrixState.setInitStack();                     //初始化变换矩阵
            ShaderManager.compileShaderHY();                //编译着色器程序
            bgd=new BackGround(ShaderManager.getTextureShaderProgram(),1,1,bg_texCoor);
            pgs_dt=new BackGround(ShaderManager.getTextureShaderProgram(),
                    0.9f,0.11f,pgs_dt_texCoor);             //进度条背景
            pgs_fgt=new Process(ShaderManager.getPrograssShaderProgram(),
                    0.9f,0.11f,pgs_fgt_texCoor,0);          //加载界面的进度条
            for(int i=0;i<no.length;i++){                   //加载界面中用到的数字纹理坐标
                float[] tempTexCoor=new float[]{
                    0.091f*i,0, 0.091f*i,1, 0.091f*(i+1),1, 0.091f*i,0, 0.091f*(i+1),
                     1, 0.091f*(i+1),0
                };
                no[i]=new BackGround(ShaderManager.getTextureShaderProgram(),
                        0.06f,0.06f,tempTexCoor);           //加载界面中用到的数字对象
            }
            backGroundId=initTextureFromBitmap(bmbackGround);//调用初始化纹理 id 的方法
            pgs_dt_id=initTextureFromBitmap(bmPgsDt);       //获取进度条背景纹理 id
            pgs_fgt_id=initTextureFromBitmap(bmPgsFgt);     //获取进度条纹理 id
            no_texId=initTextureFromBitmap(bmNum);          //获取数字图片纹理 id
    }}
```

- 第1-5行为实现GLSurfaceView.Renderer接口后重写的onSurfaceChanged方法，该方法在游戏发生改变时调用，该方法中主要是设置视窗并计算了GLSurfaceView的宽高比。
- 第6-30行为实现GLSurfaceView.Renderer接口后重写的onSurfaceCreated方法，该方法在游戏开始时调用，该方法中主要是打开背面剪裁、深度检测、编译着色器程序、创建加载界面的对象以及获取相应的纹理坐标。

（8）最后开发准备3D物体数据、准备树木的数据以及树木的方法，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的MyGLSurfaceView.java。

```java
public void tdYb(){                                         //预备 3D 物体数据的方法
    tdCount=0;
    for(int i=0;i<tdObjectList.size();i++){                 //遍历存储 3D 物体的集合
        int rowM=tdObjectList.get(i).rows-bRow;
        int colM=tdObjectList.get(i).cols-bCol;
        if(rowM>=NUMBER_MAP||rowM<=-NUMBER_MAP||
                colM>=NUMBER_MAP||colM<=-NUMBER_MAP){
            continue;                                       //不在规定的行或者列中，执行下一个
        }
            if(ClipGrid.CLIP_MASK[directNo][rowM+2][colM+2]){
                tdHz[tdCount]=i;                            //在规定的行与列中，记录该 3D 物体
                tdCount++;
} } }
……//此处省略了绘制 3D 物体的方法，读者可以自行查阅随书光盘中的源代码
List<ShrubForControl> treeListTemp=new ArrayList<ShrubForControl>();
public void treeYb(){                                       //预备树木数据的方法
    treeListTemp.clear();
    for(int i=0;i<treeList.size();i++){                     //循环列表
        int rowM=treeList.get(i).rows-bRow; int colM=treeList.get(i).cols-bCol;
        if(rowM>=NUMBER_MAP||rowM<=-NUMBER_MAP||
          colM>=NUMBER_MAP||colM<=-NUMBER_MAP){             //进行 25 宫格循环判断
            continue;                                       //不在规定的行或者列中，执行下一个
        }
        if(ClipGrid.CLIP_MASK[directNo][rowM+2][colM+2]){
            treeListTemp.add(treeList.get(i)); //在规定的行与列中，将该树木存在集合中
} } }
public void drawTrees(int dyFlag){
    if(dyFlag==1){                                          //进行远近排序
        Collections.sort(treeListTemp);
    }
    for(int i=0;i<treeListTemp.size();i++){                 //遍历列表集合绘制树木
      ss.drawSelf(treeListTemp.get(i).texIds,treeListTemp.get(i).id,
      treeListTemp.get(i).xoffset,
            treeListTemp.get(i).yoffset,treeListTemp.get(i).zoffset, dyFlag
); } }
```

- 第1-13行为准备3D物体数据的方法，主要是遍历3D物体的集合判断各个物体的行与列是否在自己使用的赛艇所在行与列周围的25个格子中，如果在，则将该数据添加到数组中；否则跳过本次循环执行下一次循环。
- 第15行为创建的存储树木控制类的集合。
- 第16-26行为准备树木数据的方法，与准备3D物体数据的方法相似，均是通过25宫格判断的。
- 第27-34行为绘制树木的方法，在该方法开始之前，首先需要将树木进行排序，然后遍历排序后的集合绘制树木。

20.7.21　运动监听线程KeyThread

前一小节介绍了本案例中的游戏界面相关类的开发，接下来本小节将开发本案中的运动监听线程——KeyThread，该线程的主要作用是判断赛艇是否与赛道碰撞、判断是否跑完一圈以及判断是否破纪录等，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/com/bn/st/clp目录下的KeyThread.java。

```
package com.bn.clp;                              //声明包名
……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
public class KeyThread extends Thread{           //运动监听线程
    MyGLSurfaceView mgmv;
    ……//此处省略了本类中部分成员变量的声明，读者可以自行查阅随书光盘中的源代码
```

```
    int dqCount=0;                               //氮气计数器
    ……//此处省略了本类中的构造器，读者可以自行查阅随书光盘中的源代码
    ……//此处省略了重写的 run 方法，读者可以自行查阅随书光盘中的源代码
    public boolean isPZ(float bx,float bz){       //判断是否与桥墩发生碰撞的方法
            float bPointX=(float) (bx-BOAT_UNIT_SIZE*Math.sin(Math.toRadians(sight_
            angle)));
            float bPointZ=(float) (bz-BOAT_UNIT_SIZE*Math.cos(Math.toRadians(sight_
            angle)));
            float carCol=(float) Math.floor((bPointX+UNIT_SIZE/2)/UNIT_SIZE);
            float carRow=(float) Math.floor((bPointZ+UNIT_SIZE/2)/UNIT_SIZE);
            for(PZZ temp:ma.gameV.pzzList){
                if(temp.row==carRow&&temp.col==carCol){ //桥与赛艇均在同一行、列中
                    if(temp.isIn(bPointX, bPointZ)){
                        return true;        //碰撞返回 true，否则返回 false
            } } }
            return false;
        }
        ……//此处省略了判断赛艇是否与河道碰撞的方法，读者可以自行查阅随书光盘中的源代码
        final float RACE_HALF_X=14*UNIT_SIZE; final float RACE_HALF_Z=20*UNIT_SIZE;
        final float RACE_BEGIN_X=30; final float RACE_BEGIN_Z=90;
        public void isHalfForBoat(float carTempX,float carTempZ){
            double dis=Math.sqrt((carTempX-RACE_HALF_X)*(carTempX-RACE_HALF_X)
                    +(carTempZ-RACE_HALF_Z)*(carTempZ-RACE_HALF_Z));
            if(dis<=120){
                halfFlag=true;               //赛艇跑完半圈，则设置 halfFlag 为 true
        } }
        public void isOneCycleForBoat(float carTempX,float carTempZ){//是否跑完一圈的方法
            double dis=Math.sqrt((carTempX-RACE_BEGIN_X)*(carTempX-RACE_BEGIN_X)
                +(carTempZ-RACE_BEGIN_Z)*(carTempZ-RACE_BEGIN_Z));
            if(dis<=60&&bOldZ>RACE_BEGIN_Z&&carTempZ<=RACE_BEGIN_Z){
                if(halfFlag==true){
                    numberOfTurns=numberOfTurns+1; //圈数加 1
                    if(numberOfTurns==3&&isSpeedMode){       //跑完竞速模式
                        RANK_FOR_HERO_BOAT=RANK_FOR_HELP;
                        CURR_BOAT_V=0;               //设置速度为 0
                        BOAT_A=0;                    //设置加速度为 0
                        String currSysTime=Constant.getCurrTime();
                        String currUseTime=Constant.getUseTime();
                        DBUtil.insertRcRDatabase(currSysTime,
                                currUseTime, RANK_FOR_HERO_BOAT);
                        ma.hd.sendEmptyMessage(12);//弹出对话框
                    }else if(numberOfTurns==3&&!isSpeedMode){   //跑完计时模式
                        CURR_BOAT_V=0;               //设置速度为 0
                        BOAT_A=0;                    //设置加速度为 0
                        String currSysTime=Constant.getCurrTime();
                        String currUseTime=Constant.getUseTime();
                        List<String> alist=DBUtil.getTimeFromJSDatabase();
                        isBreakRecord=isFast(currUseTime,alist);
                        DBUtil.insertJSDatabase(currSysTime, currUseTime);
                        ma.hd.sendEmptyMessage(11);//弹出是否破纪录的对话框
                } }
                halfFlag=false;                      //设置该标志位为 false
```

```
56            }else if(dis<=60&&bOldZ<=RACE_BEGIN_Z&&carTempZ>RACE_BEGIN_Z){
57                halfFlag=false;
58        } }
59        public static boolean isFast(String currTime,List<String> aList){//是否破纪录方法
60            boolean result=false;
61            String[] str=currTime.split(":");                    //切割字符串
62            int currT=Integer.parseInt(str[0])*60*100+Integer.parseInt(str[1])*
              100+Integer.parseInt(str[2]);
63            List<Integer> tempInteger=new ArrayList<Integer>();
64            for(int i=0;i<aList.size();i++){                     //遍历数据集合
65                String[] tempStr=aList.get(i).split(":");   //切割字符串
66                tempInteger.add(Integer.parseInt(tempStr[0])*60*100+
67                        Integer.parseInt(tempStr[1])*100+Integer.parseInt(tempStr[2]));
68            }
69            Collections.sort(tempInteger);
70            if(tempInteger.size()==0){                    //如果数据库中没有数据
71                result=true;
72            }else if(currT<tempInteger.get(0)){      //当前数据小于数据库中的最小数据
73                result=true;
74            }
75            return result;
76    } }
```

- 第9-20行为本类中判断是否与桥墩发生碰撞的方法，首先判断赛艇是否与所有桥中的某一个在同一个格子内，如果在，则通过PZZ对象调用isln方法判断是否发生碰撞；否则，则返回false。
- 第24-29行为判断赛艇是否已经跑了半圈，如果跑了半圈，则设置halfFlag为true，否则设置halfFlag为false。
- 第30-58行为判断赛艇是否跑完一圈的方法。如果跑完一圈，则将圈数加1，并判断游戏圈数是否为3。如果圈数为3，说明本局游戏已经结束，需要根据游戏的模式将游戏中的速度以及加速度等值为0，并且发送不同的消息弹出相应的对话框。
- 第59-76行为判断游戏在计时模式下是否打破纪录的方法。主要是去除数据库中的数据，将数据转换为int类型的值并添加到相应的集合中，之后通过Collections调用sort方法将存储数据库数据的集合进行排序获取最小值。最后判断本局用时是否小于数据库中的最小记录，小于则认为打破纪录。如果数据库中根本没有数据，那么不需要比较直接返回true。

20.8　该游戏中的着色器

前一节已经介绍了游戏界面相关类的开发，接下来本小节将开发本游戏中使用到的着色器。由于前面章节中已经涉及了部分着色器，所以本小节将介绍本游戏中使用到的部分着色器的代码开发。

（1）首先开发根据过程纹理技术开发的顶点着色器，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/assests目录下的vertex_b_yz.java。

```
uniform mat4 uMVPMatrix;                    //总变换矩阵
attribute vec3 aPosition;                   //顶点位置
attribute vec2 aTexCoor;                    //顶点纹理坐标
```

```
varying vec2 vTextureCoord;                 //用于传递给片元着色器的变量
varying float currX;                        //当前片元的x坐标
void main(){
    gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点位置
    vTextureCoord = aTexCoor;               //将接收的纹理坐标传递给片元着色器
    currX=aPosition.x;
    }
```

说明

上述顶点着色器中主要是根据总变换矩阵计算此次绘制此顶点位置，将接收的纹理坐标传递给片元着色器，同时将顶点Y坐标通过易变变量传递给片元着色器。

（2）完成了过程纹理的顶点着色器之后，接下来开发过程纹理的片元着色器，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/assests目录下的frag_b_yz.java。

```
precision mediump float;
varying vec2 vTextureCoord;                    //接收从顶点着色器过来的参数
uniform sampler2D sTextureGrass;               //纹理内容数据(草皮)
uniform sampler2D sTextureRock;                //纹理内容数据(岩石)
uniform float b_YZ_StartX;                     //陆地起始X
uniform float b_YZ_XSpan;                      //陆地X偏移量
varying float currX;                           //当前片元的片元x值
void main(){
    float min=0.2;                             //过渡区间开始值
        float max=0.6;                         //过渡区间结束值
        float currXRatio=(currX-b_YZ_StartX)/b_YZ_XSpan;
        vec4 gColor=texture2D(sTextureGrass, vTextureCoord);     //草地颜色的四维向量
        vec4 rColor=texture2D(sTextureRock, vTextureCoord); //岩石颜色的四维向量
        vec4 finalColor;                       //最终颜色值
        if(currXRatio<min){                    //当前片元的x值小于过渡区间开始值
            finalColor=gColor;
        }else if(currXRatio>max){              //当前片元的x值大于过渡区间开始值
            finalColor=rColor;
        }else{
            float rockBL=(currXRatio-min)/(max-min);
            finalColor=rockBL*rColor+(1.0-rockBL)*gColor;
        }
        gl_FragColor = finalColor;             //给此片元从纹理中采样出颜色值
    }
```

- 上述片元着色器中不再是仅采用草皮纹理对山体进行着色，而是同时采用了两幅纹理。
- 当片元Y坐标小于过程纹理起始Y坐标时用草皮纹理着色，大于过程纹理起始Y坐标加跨度时采用岩石纹理着色。位于二者之间时将草皮纹理、岩石纹理按照计算的比例进行混合，混合的规则为片元Y坐标越大，岩石纹理所占百分比越大。

（3）接下来开发进度条的顶点着色器，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/assests目录下的vertex_prograss.java。

```
uniform mat4 uMVPMatrix;                //总变换矩阵
```

```
attribute vec3 aPosition;               //顶点位置
attribute vec2 aTexCoor;                //顶点纹理坐标
varying vec2 vTextureCoord;             //用于传递给片元着色器的变量
varying vec3 vPosition;
void main(){
    gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点位置
    vTextureCoord = aTexCoor;           //将接收的纹理坐标传递给片元着色器
    vPosition=aPosition;
    }
```

说明

上述顶点着色器中主要是根据总变换矩阵计算此次绘制此顶点位置，将接收的纹理坐标传递给片元着色器，同时将顶点坐标通过易变变量传递给片元着色器。

（4）完进度条的顶点着色器的开发之后，接下来将开发进度条的片元着色器，具体代码如下。

代码位置：见随书光盘中源代码/第20章/BNST/assests目录下的frag_prograss.java。

```
precision mediump float;
varying vec2 vTextureCoord;              //接收从顶点着色器过来的参数
uniform sampler2D sTexture;              //纹理内容数据
uniform float aPrograss;
varying vec3 vPosition;                  //片元的坐标
void main(){
    if(vPosition.x<aPrograss){
        gl_FragColor = texture2D(sTexture, vTextureCoord);//给此片元从纹理中采样出颜色值
    }else{
            discard;
    } }
```

说明

上述片元着色器主要是根据顶点着色器通过易变变量传入到片元着色器中的坐标与直接传入到片元着色器的进度值作比较，如果传入的顶点坐标小于进度值，则给此片元从纹理中采样出颜色值，否则调用discard直接丢弃。

20.9　游戏的优化与改进

至此，本案例的开发部分已经介绍完毕。本游戏基于OpenGL ES 2.0渲染引擎，笔者还对本游戏做了许多优化，但是本游戏仍然还存在许多的不足之处。

- 游戏界面的改进。

本游戏界面虽然已经十分漂亮，但是还有一定的改进空间，读者可以根据自己的需求更改界面图片。

- 竞速模式中赛艇自动开。

在竞速模式中除了自身使用的赛艇外，其他赛艇都是自动开的，但是本游戏中自动开的赛艇路径是固定的，并且不能更改赛艇的速度，读者可以将赛艇自动开更改的更加智能。

- 游戏的进一步优化。

虽然笔者已经将本游戏中必须优化的部分优化完毕，但是一些机器运行还是比较慢，因此读者可以将本游戏再进一步地优化，使得游戏更加流畅。

第21章

火力篮球

随着移动互联网的快速发展，智能手机的进一步完善。人们在移动设备上进行娱乐的时间越来越多，而随着在智能手机上的模拟现实技术越来越真实化，各种模拟现实类游戏更是发展迅速。

本章将开发一款基于Android平台上模拟现实的投篮游戏——火力篮球，该游戏采用第三方物理引擎模拟实现了现实中的投篮游戏，下面对该游戏的开发进行介绍。

21.1　游戏背景及功能概述

本小节将对火力篮球游戏进行简单的介绍，通过本小节的学习，可以使读者对本游戏有一个整体的认识，了解本章将要开发案例的具体功能及效果。

21.1.1 背景概述

火力篮球模拟了现实中的投篮游戏机，而投篮游戏机又是源于街头篮球，街头篮球起源于美国，现在已是流行于世界各地的体育竞技项目。本游戏就是将这一经典的体育运动带到手机中，在该游戏中只需动一下手指即可享受投篮游戏的乐趣。

火力篮球是借助第三方物理引擎JBullet，使用OpenGL ES 2.0进行图像渲染，结合智能手机的触摸技术打造的一款小型手机游戏。玩家通过触摸屏幕，单击球并且移动手指，程序通过玩家移动的方向和距离来换算投篮的方向和初始速度。该游戏用到的相关技术如下所列。

- JBullet物理引擎：是一个开源的物理模拟计算引擎，世界三大物理模拟引擎之一（另外两种是Havok和PhysX）。
- OpenGL ES 2.0：是OpenGL三维图形API的子集，针对手机、PDA和游戏主机等嵌入式设备而设计。该API由Khronos集团定义推广，Khronos是一个图形软硬件行业协会，该协会主要关注图形和多媒体方面的开放标准。

21.1.2 功能介绍

本小节将对火力篮球游戏的功能以及操作方法进行简单的介绍，经过这一节的学习，读者应该对该游戏有一定的感性认识，其具体的操作步骤如下。

（1）首先运行游戏，进入游戏主菜单界面，其效果如图21-1所示。



图21-1　主菜单界面

（2）在菜单界面中，单击开始按钮，进入的是游戏加载界面，其效果如图21-2所示。当加载 完毕后，直接进入游戏界面。在该界面下，玩家通过触摸一个静止的球并且移动一定距离，程序根据移动方向和移动的距离给予初速度，进一个球加一分，总时间为一分钟，其效果图如图21-3和图21-4所示。

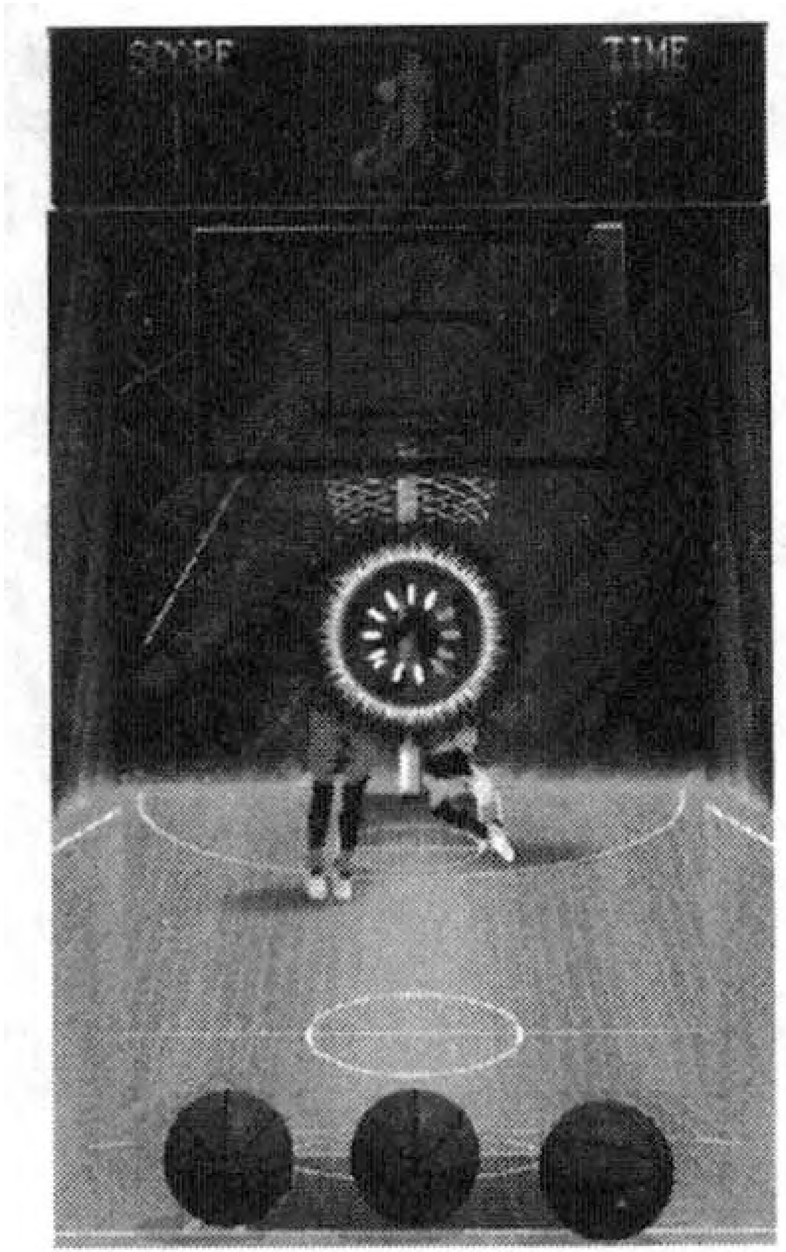


图21-2　加载界面

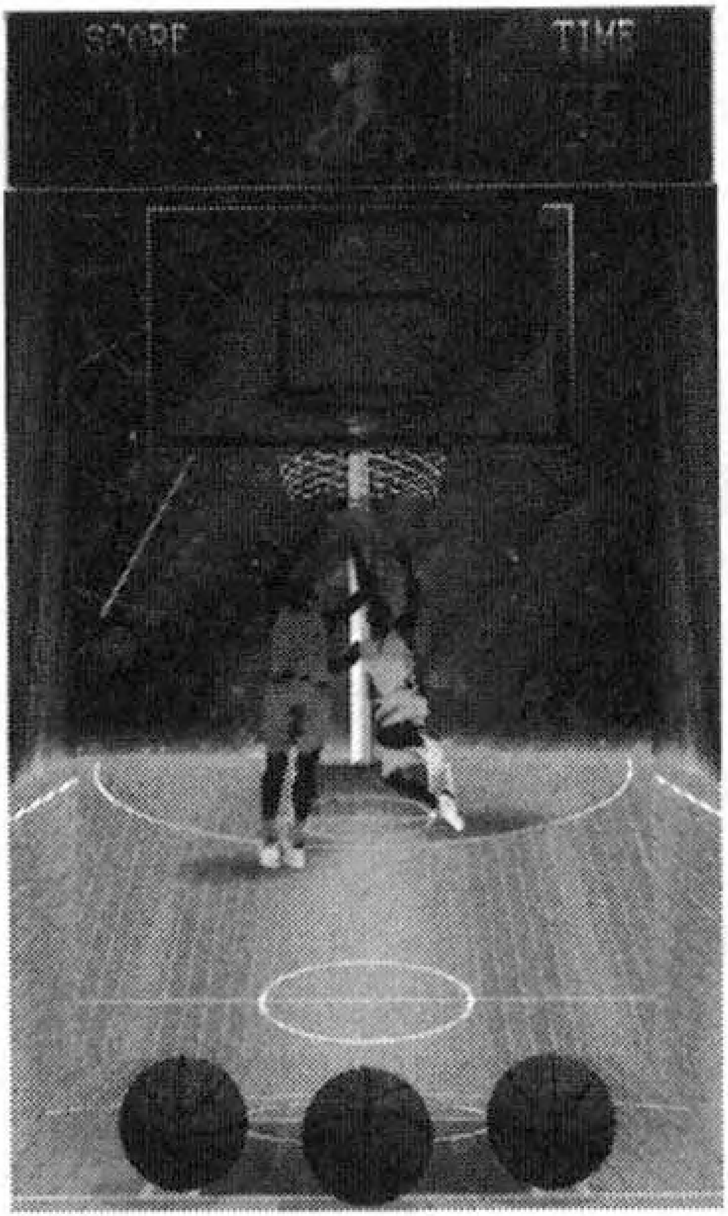


图21-3 游戏界面（1）

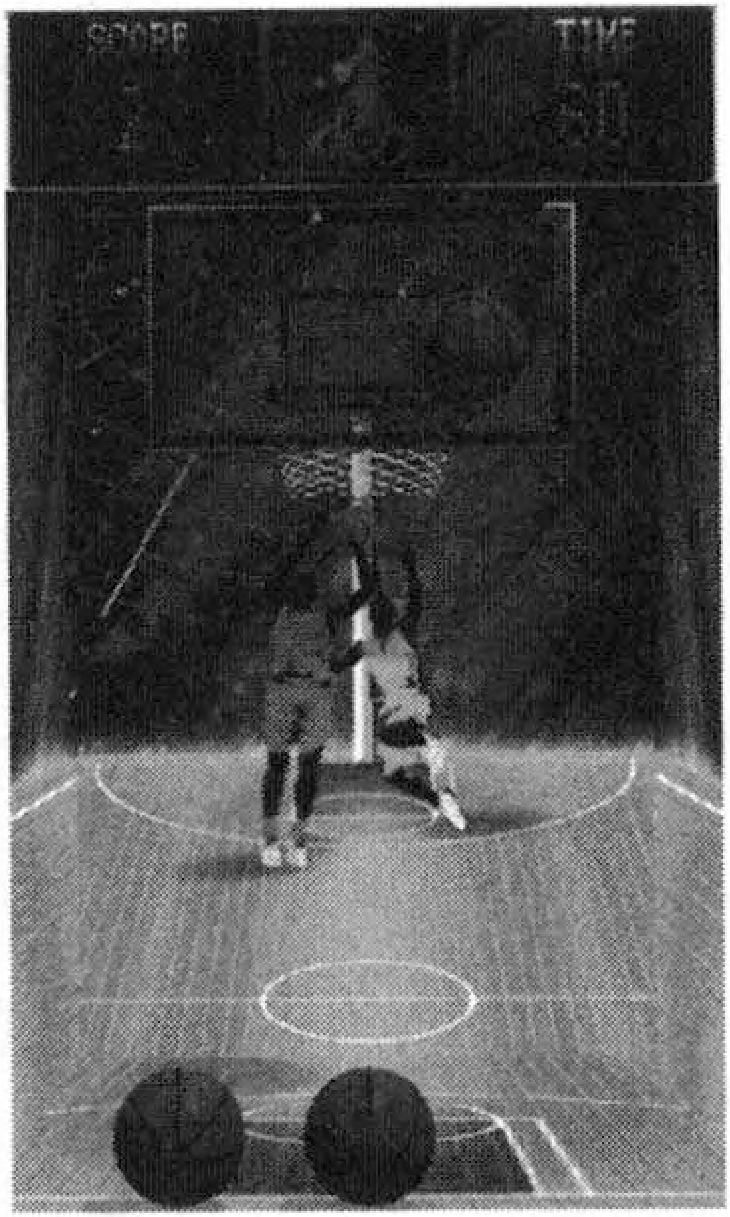


图21-4　游戏界面（2）

（3）当游戏时间到达一分钟时游戏结束，游戏界面将自动切换到游戏结束界面，其效果如图21-5所示，在该界面下可单击再来一次按钮继续游戏，也可单击返回菜单按钮返回到主菜单，还可以单击退出按钮直接退出游戏。

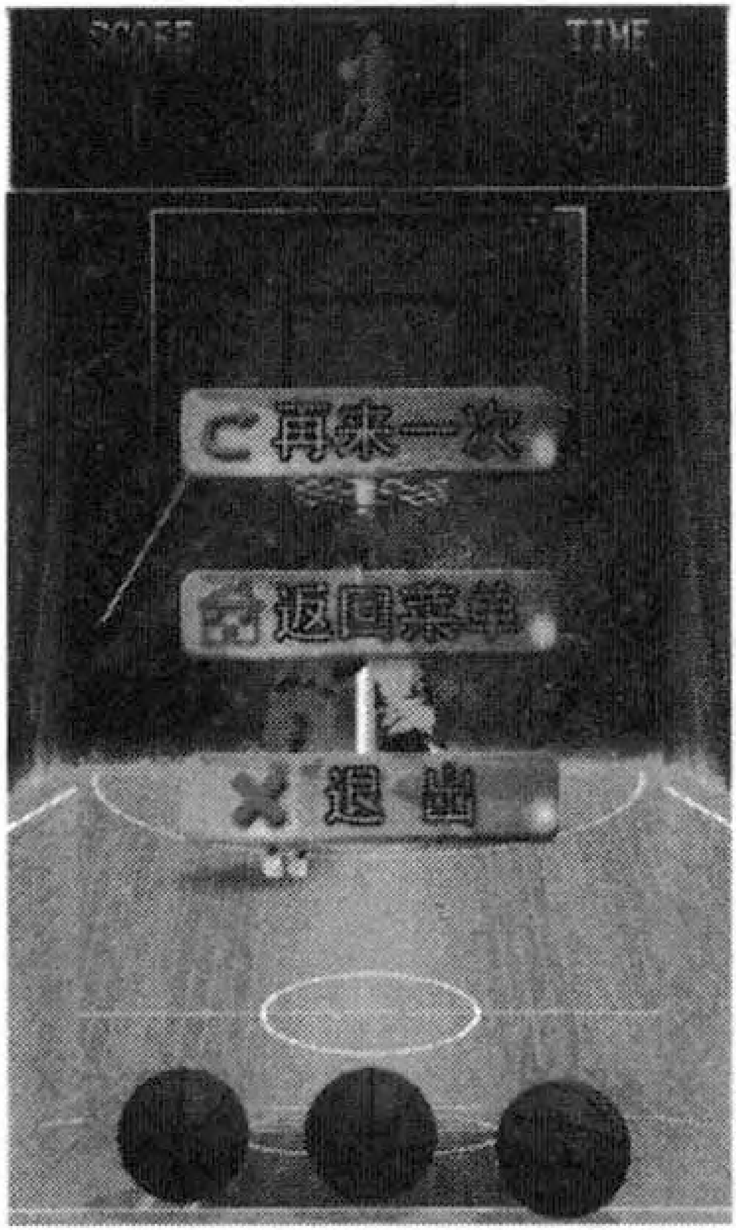


图21-5　游戏结束界面

（4）主菜单界面下，单击设置按钮，即可进入设置界面，对背景音乐和场景音乐进行设置。其效果如图21-6所示。

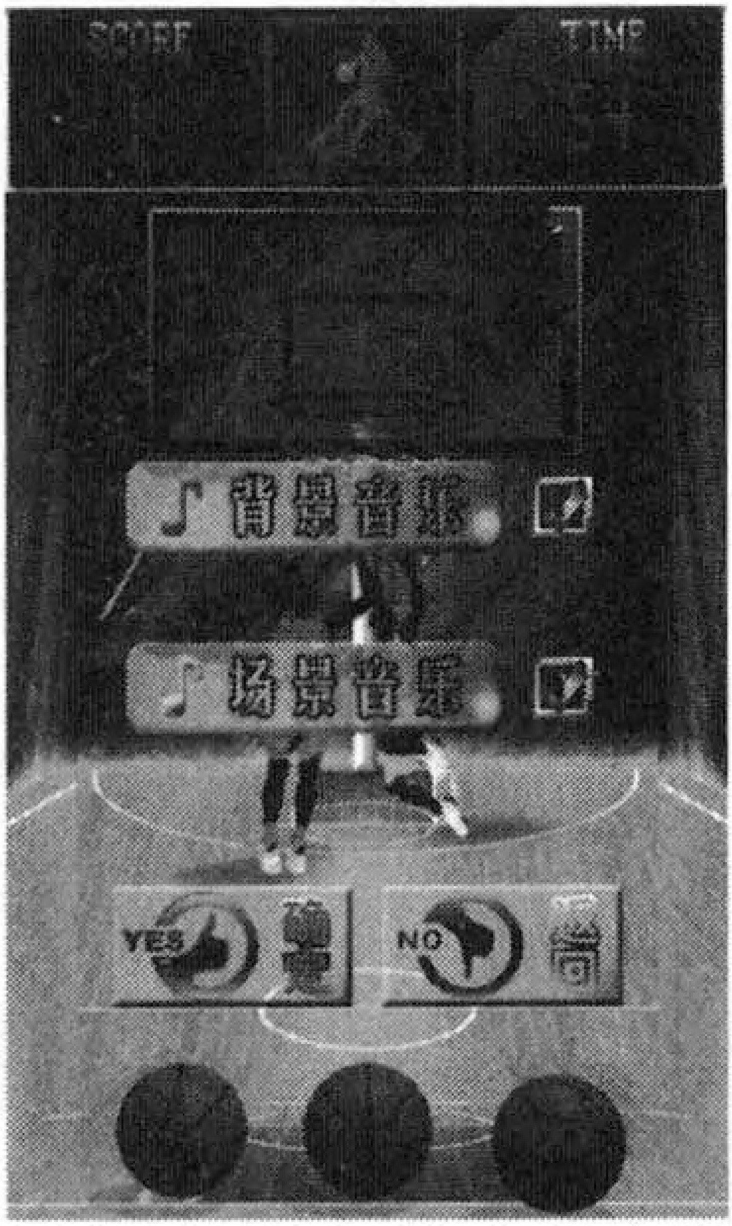


图21-6　设置界面

（5）主菜单界面下，单击关于按钮，进入关于界面，其效果如图21-7所示。



图21-7　关于界面

（6）主菜单界面下，单击帮助按钮将首先进行游戏界面加载，然后进入帮助界面，在该界面下可以单击右下角的暂停播放按钮实现暂停播放，画面不再变化，此时单击播放按钮可继续播放，而单击停止按钮则直接跳到游戏界面，帮助界面的效果图如图21-8和图21-9所示。

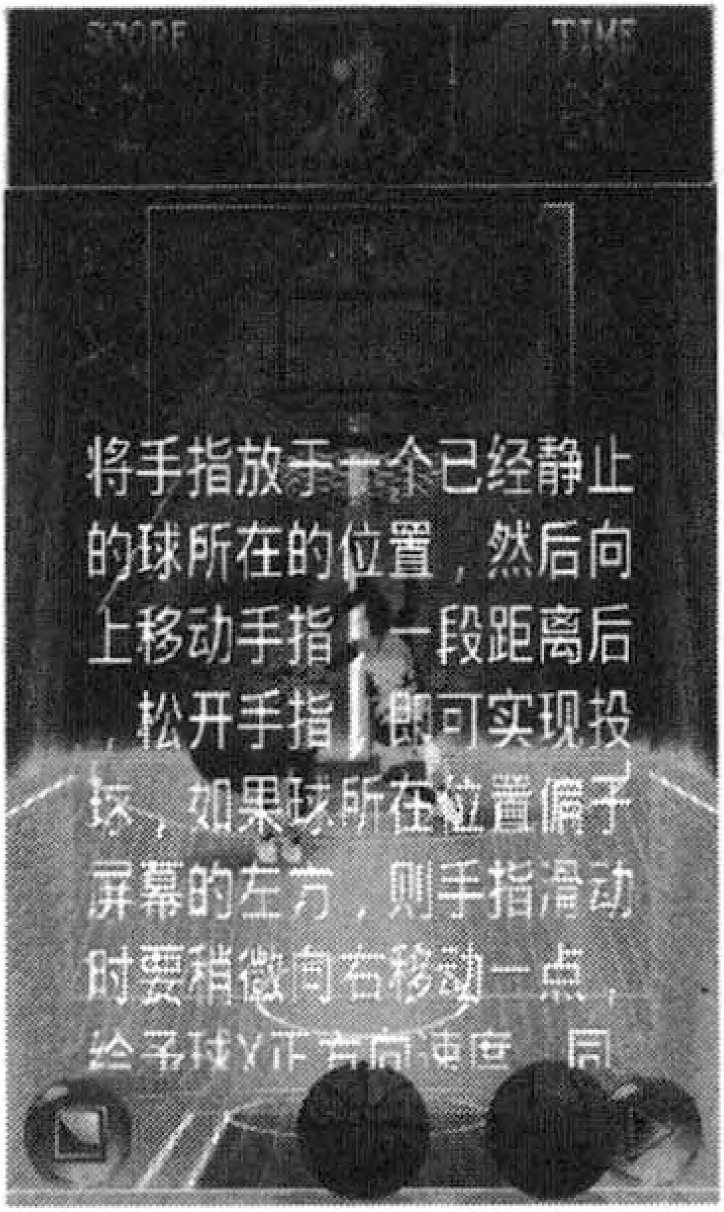

将手指放于一个已经静止
的球所在的位置，然后向
上移动手指一段距离后
，松开手指，即可实现投
球，如果球所在位置偏于
屏幕的左方，则手指滑动
时要稍微向右移动一点，

图21-8　帮助界面（1）

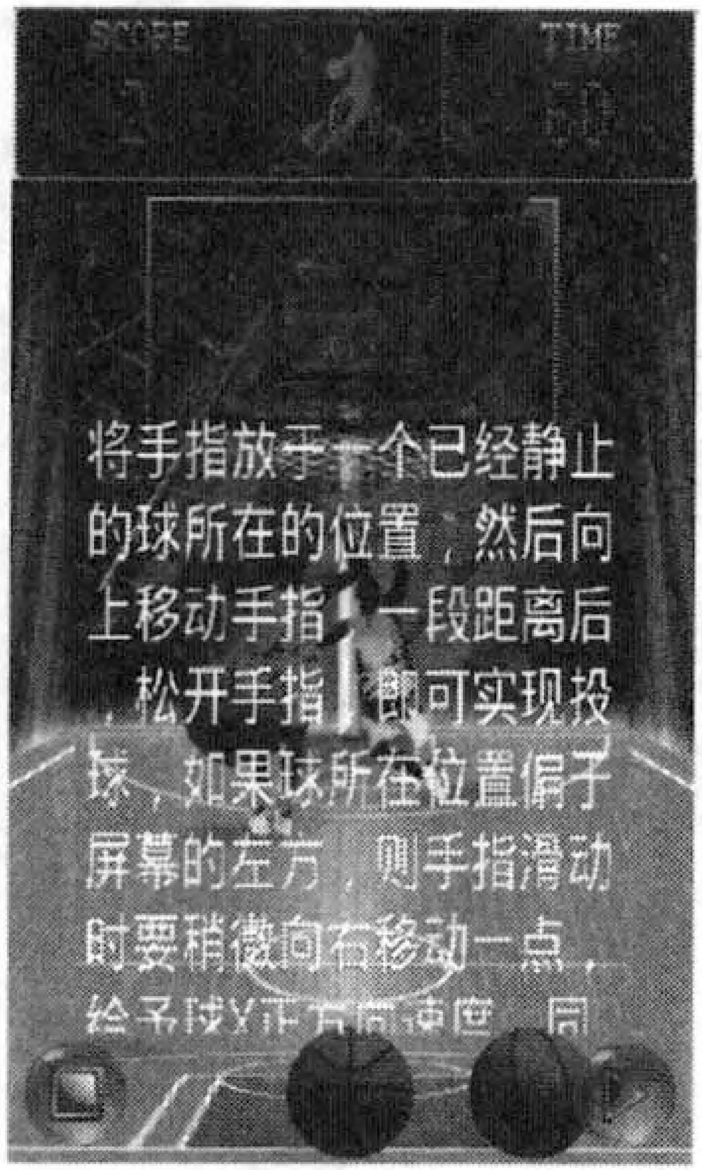


图21-9　帮助界面（2）

（7）主菜单界面下，单击记录按钮，进入排行榜界面，在该界面下有两种排序方式，一种按照 时间排序，另一种按照成绩排序，单击上方的TIME和GRADE按钮即可查看按相应方式排序的游戏成绩。其效果如图21-10和图21-11所示。

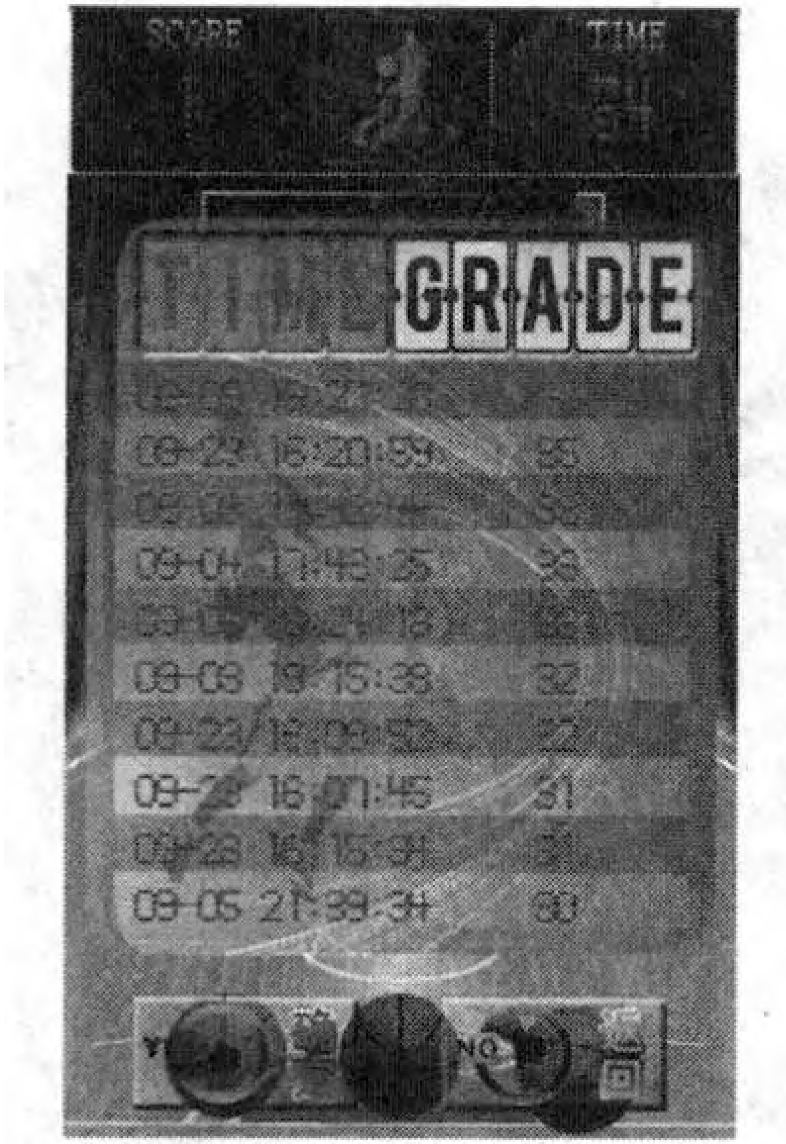


图21-10　按成绩排序

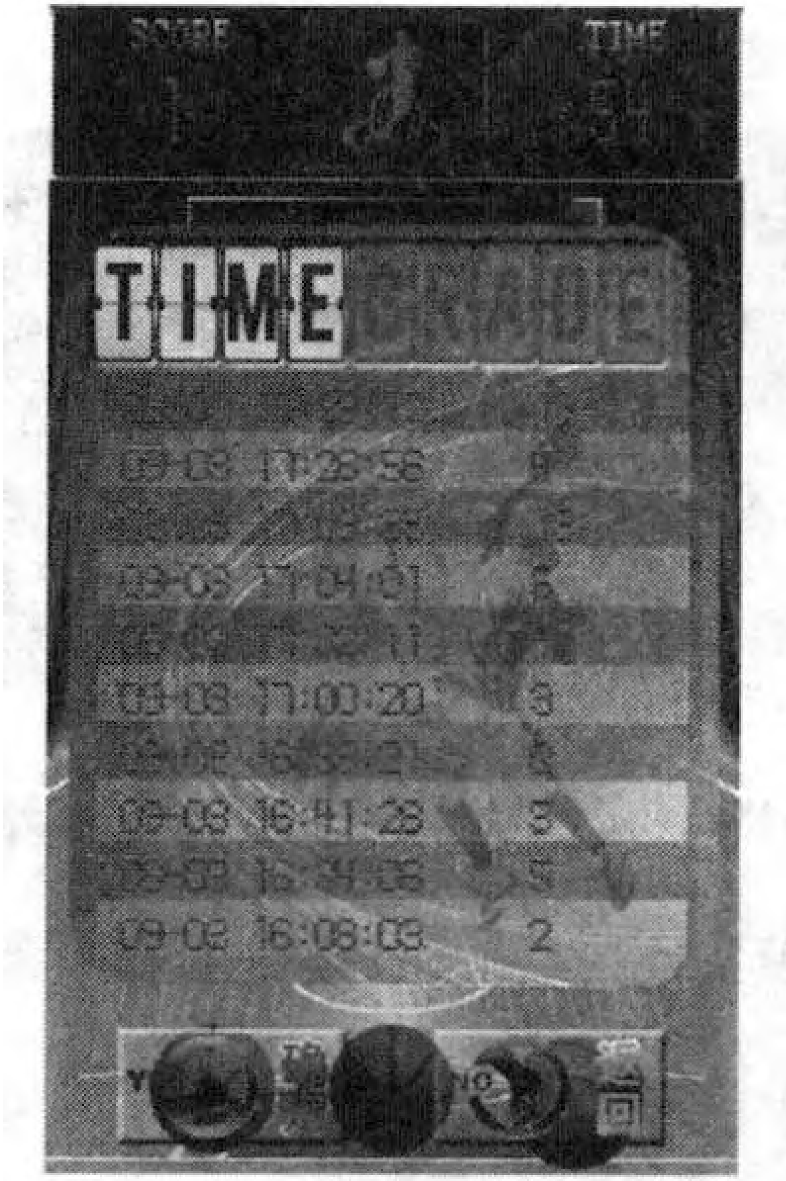


图21-11　按时间排序

21.2　游戏的策划及准备工作

前一小节对游戏的背景和相应的游戏功能进行了简单的介绍，本小节将对游戏的整体策划及开发前的一些准备工作进行介绍，主要包括游戏风格的设计、程序中用到的各种资源的收集和整合。这个环节在整个游戏开发中是至关重要的，不可忽视。

21.2.1 游戏的策划

下面对该游戏的策划进行简单的介绍。真实的游戏项目开发中，游戏的策划会非常的细致、具体、全面。此处由于篇幅所限，仅对该步骤进行了简单的说明。该游戏的策划如下所列。

- 游戏类型。

该游戏应用第三方物理引擎JBullet，以OpenGL ES 2.0作为游戏场景呈现技术，属于休闲类游戏。

- 运行目标平台。

游戏目标平台为Android 2.2及其以上版本，由于使用OpenGL ES 2.0技术，所以必须在存在显卡的Android设备上运行。

- 目标受众。

火力篮球操作简单、思维灵活性强、采用移动设备为载体、整个游戏真实感强，适合老年，儿童以及追求时尚的青年人。

- 操作方式。

玩家触摸一个静止的球，并且向一个方向滑动，程序根据滑动的方向、滑动的距离和触摸的球，给予这个球一个初速度。然后便是程序对整个游戏场景实行物理模拟，如果此次投篮成功则加一分，否则不计分。

- 呈现技术。

本游戏使用OpenGL ES 2.0作为呈现技术，对于篮球的阴影则是采用动态阴影技术，增加了游 戏场景的真实感。

21.2.2 Android平台下游戏开发的准备工作

完成游戏策划的介绍后，下面需要做一些游戏开发前的准备工作，主要包括搜集和制作图片、声音等资源，以及将JBullet物理引擎的支持jar包准备好等，其详细步骤如下。

（1）首先介绍的是该游戏中所有用到的图片资源，将所有图片资源都放在项目文件下的res/drawable-nodpi文件夹中，其详细情况如表21-1所列。

表21-1　图片清单

图片名	大小（KB）	像素（w×h）	用途	图片名	大小（KB）	像素（w×h）	用途
about1.png	10.1	250×60	关于按钮	guanyuwenzitu.png	264	450×520	关于界面的文字图片
background.jpg	62.8	480×800	主菜单背景	gundontiao.png	2.74	8×50	排行榜的滚动条
baisefangge.png	3.93	40×40	声音设置界面中的方框	help1.png	19.5	250×60	帮助按钮
baisewaikuang.png	109	450×520	关于界面的背景	hengxian.png	2.94	16×30	横线
bangzuwelcome.jpg	71.2	512×1024	加载界面的背景	honsegou.png	4.19	40×40	标记对应声音是否开启的钩
basketball.jpg	36.3	128×128	篮球纹理	icon.png	9.15	64×64	应用程序图标
basketnet.png	105	256×256	篮网纹理	jilu.png	21.8	250×60	记录按钮
begin.png	26.2	250×60	开始游戏按钮	lanban.png	15.1	320×320	篮板纹理
beijingyinyu.png	23.2	250×60	背景音乐文字	maohao.png	2.91	16×30	冒号
changjinyinyu.png	24.0	250×60	场景音乐文字	number.png	21.9	256×32	数字
d0.png	3.58	16×30	数字 0	pause.png	8.84	64×64	暂停按钮
d1.png	3.25	16×30	数字 1	play.png	9.36	64×64	播放按钮
d2.png	3.5	16×30	数字 2	quedinganniu.png	15.4	160×80	确定按钮
d3.png	3.53	16×30	数字 3	retry.png	23.8	250×60	再来一次按钮
d4.png	3.51	16×30	数字 4	shezhi.png	20.1	250×60	设置按钮
d5.png	3.57	16×30	数字 5	shijianbeijing2.png	176	450×520	排行榜按时间排序背景
d6.png	3.63	16×30	数字 6	shou.png	16.3	128×128	帮助界面下的手
d7.png	3.25	16×30	数字 7	shut.png	19.5	250×60	退出按钮
d8.png	3.65	16×30	数字 8	stop.png	8.84	64×64	停止按钮

续表

图片名	大小（KB）	像素（w×h）	用途	图片名	大小（KB）	像素（w×h）	用途
d9.png	3.57	16×30	数字9	swall1.jpg	122	256×512	左面墙纹理
dialog.9.png	2.0	150×88	对话框背景	swall2.jpg	165	256×512	后面墙纹理
dott.png	305	512×512	加载界面选择的图片	swall3.jpg	122	256×512	右面墙纹理
fanhuianniu.png	14.8	160×80	返回按钮	timegradebeijing.png	179	450×520	排行榜按分数排序时背景
fanhuicaidan.png	25.0	250×60	返回菜单按钮	yibiaoban.png	17.9	320×64	仪表板背景
floor.png	22.2	128×128	地板的纹理				

（2）本游戏用到的声音资源存放在项目目录中的res/raw文件夹中，其详细情况如表21-2所列。

表21-2　声音清单

声音文件名	大小（KB）	格式	用途
beijingyingyu.mp3	233	mp3	背景音乐
levelend.ogg	20	ogg	游戏时间结束声音
pengzhuang.ogg	14	ogg	碰地板、墙等声音
shoot.ogg	7	ogg	进球声音

（3）本游戏用到的构造篮筐模型的obj文件存放在项目目录中的assets文件中。其详细情况如表21-3所列。

表21-3　模型文件清单

文件名	大小（KB）	格式	用途
lankuang.obj	25.8	obj	篮筐3D模型文件

（4）完成图片、声音等资源的准备后，还要将JBullet物理引擎的两个支持jar包jbullet.jar与vecmath.jar导入到本章项目的lib目录下。

21.3　游戏的架构

该游戏代码的开发之前，首先需要对游戏的框架进行简单的介绍，读者从中能更好地理解游戏的开发过程，从而更加容易理解后面章节中介绍的内容。

21.3.1　各个类简要介绍

为了让读者更好地理解各个类的作用，下面将分成3部分对这些类进行简单介绍。对于各个类的详细代码将在后面的章节中相继给出。

1．公共类

Activity的实现类BasketBall_Shot_Activity，该类通过扩展Activity得到，为整个游戏的控制器，也是整个游戏的程序入口。

2．辅助界面相关类

- 主菜单界面类CaiDanView。

主菜单界面类CaiDanView为游戏菜单界面的实现类，负责绘制菜单界面以及对菜单界面屏幕进行监听。根据不同触控时间进行界面的切换。

- 游戏设置界面类ShengyinKGJiemian。

游戏设置界面类ShengyinKGJiemian是对游戏中对应声音进行相关设置的类。负责绘制设置界面中的相关内容和对界面屏幕进行监听。

- 关于界面类GuanYuView。

关于界面对游戏的版权等方面进行了说明，GuanYuView类负责绘制关于界面。

- 记录界面类JiLuView。

记录界面即排行榜界面，JiLuView类从数据库中取出时间和得分，然后将结果绘制在该界面中。其排序方式分为两种，一种是按照时间的先后进行排序，另一种是按照玩家得分多少进行排序。

- 游戏结束界面类YouXiuJieShuView。

游戏结束界面为游戏时间到达一分钟后显示的界面。游戏结束时，玩家可以选择再来一局，返回主菜单或者退出按钮来实现相应界面跳转的操作。

- 数据库相关类SQLiteUtil。

该类为游戏得分提供数据存储的功能，包括创建数据库、插入数据和查询数据等。

3. 游戏界面相关类

- 游戏界面类GLGameView。

该类为整个游戏中最主要的类，控制整个游戏的运行，并负责三维空间中所有物品，如墙、篮筐、篮板、球、数字、仪表板等物体的绘制工作。

- 常量类Constant。

该类为整个游戏开发中提供一些静态常量、静态成员变量和静态方法。

- 篮球控制类BasketBallForDraw。

该类为篮球的绘制类，根据对应刚体获得自身位置和旋转角度绘制篮球。

- 蓝球的描述类BasketBallTextureByVertex。

该类为一个球的顶点位置、纹理坐标和法向量等的描述类。该类实现具体绘制一个球，同时还包括绘制该球在对应平面上的影子。

- 法向量记录类FaXiangLiang。

该类将一个点的法向量记录下来，在计算平均法向量时会用到。

- 欢迎界面下的纹理矩形类HuanYingJieMianJuXing。

该类绘制一个纹理矩形，包括顶点位置和纹理坐标等的描述。

- 绘制数字类HuiZhiShuZi。

该类根据传入的由数字组成的字符串，绘制时间和分数。其原理为先创建10个纹理矩形，给予不同的纹理坐标，然后根据数字来绘制对应的矩形。

- 添加胶囊刚体进物理世界类JiaoNangTianjiaBody。

该类为将胶囊刚体添加进物理世界，每构造一个该类的对象即添加一个刚体进物理世界。通过对其位置和旋转设置，通过多个胶囊来共同搭建一个篮筐。

- 篮板类LanBan。

该类为篮板的绘制类，在创建其对象时要传入篮板的长、宽和高。而其对应的刚体是用的JBullet中的Box代替的。

- 篮网类LanWang。

该类为篮网的绘制类，篮球投进篮筐时，篮网会左右扰动一下，此效果需要在着色器中实现，其原理为当篮球进球时，传入一个参

数，看是让其向左扰动还是向右扰动，顶点着色器对其坐标进行相应的变换即可。

- 模型类MoXing。

该类为用一个工具类从模型文件中加载顶点坐标、法向量和纹理坐标等，构造对应的缓冲数据，然后将该模型绘制出来，此游戏中的篮筐就是使用该类绘制的。

- 模型数据加载类MoXingJiaZai。

该类用于加载obj文件中的数据，将结果存储在对应数组中，然后将对应数据传给模型类构造器构造模型对应的缓冲数据，最后在总绘制中调用对应方法绘制出整个模型。

- Shader程序管理类ShaderManager。

该类将整个游戏中用到的所有Shader程序管理起来，这样使整个游戏开发中对这方面的资源管理就不必那么繁杂，减轻了工作量。

加载顶点着色器与片元着色器的工具类ShaderUtil。该类用于加载对应的着色器程序，包括顶点着色器和片元着色器，然后生成对应的着色器程序，供显卡使用。

说明

ShaderUtil工具类在本书第三章中已经进行过详细介绍，因此在本章中对于该类将不再进行介绍，读者可自行查阅随书光盘的源代码。

- 存储系统矩阵状态类MatrixState。

该类为存储系统矩阵状态的类，其中存储当前变换矩阵、摄像机矩阵、总变换矩阵等数据信息。

说明

MatrixState工具类在本书第4章中已经进行过详细介绍，因此在本章中对于该类将不再进行介绍，读者可自行查阅随书光盘的源代码。

- 将四元数转换为角度及转轴向量的类SYSUtil。

该类的作用为将描述对应刚体状态的四元数转换为对应坐标轴的旋转角度，在进行篮球的绘制 时会用到该类。

- 添加刚体进物理世界的类TianjiaBody。

该类的作用为添加一个刚体到物理世界中，该类在添加刚体到物理世界时，只有位置的设定，并没有旋转变换以及密度的设定。例如：该游戏中的几堵墙，密度为0处于静止状态。

- 纹理矩形类WenLiJuXing。

该类的作用为根据传入的宽度与高度数据，绘制相应高与宽的纹理矩形。

- 圆柱类Yuanzhu。

该类的作用为根据顶点数据、纹理坐标数据和法向量数据绘制一个圆柱。游戏中的篮板支架便是该类的对象。

21.3.2 游戏框架简介

本节将对该游戏的整体架构进行介绍，使读者对本游戏有更深的理解，其框架如图21-12所示。



图21-12　游戏框架图

说明

该游戏的框架主要分为3个部分：公共类、辅助界面相关类、游戏界面相关类。其中公共类为继承自Activity的BasketBall_Shot_Activity类。辅助界面相关类主要为关于界面、记录成绩的界面等相关界面。游戏界面相关类中包括一些游戏实体和工具类。

下面按照程序运行的顺序介绍各个类的作用以及该游戏的整体框架，其详细步骤如下。

（1）启动游戏，首先运行的是BasketBall_Shot_Activity类，在该类中根据相应代码切换到主菜单界面，从而运行CaiDanView类。

（2）主菜单界面中，当玩家单击开始游戏按钮时，将创建GLGameView类的对象。单击开始游戏按钮后，界面首先切换到游戏加载界面，加载界面为一个3D界面。此处设计游戏加载界面的作用为防止出现系统资源加载过程中的黑屏效果。

（3）游戏界面中，玩家触摸球并给予其相应的速度，在一分钟内进球个数为玩家在该局的成绩。当游戏结束后，界面切换到游戏结束界面YouXiuJieShuView，在该界面中存在3个按钮，分别为 再来一次按钮、返回主菜单按钮和退出按钮，玩家可根据需要单击其中某个按钮进行相应的操作。

（4）主菜单界面中，玩家单击设置按钮时，进入相关设置界面ShengyinKGJiemian，在该界面中可进行是否开启背景音乐和场景音乐的设置。

（5）主菜单界面中，玩家单击关于按钮时，进入关于界面GuanYuView，也可选择帮助按钮，进入帮助界面。此处的帮助界面其实和游戏界面是同一个界面，在帮助界面中，为玩家演示了如何投篮的动作，由程序自动完成，玩家不可操控。

（6）主菜单界面中，玩家单击记录按钮时，进入排行榜界面JiLuView。该界面中玩家的成绩存在两种排序方式：按照成绩排序、按照时间排序。

（7）最后在主菜单界面下可单击退出按钮退出游戏。

21.4 公共类BasketBall_Shot_Activity

从本节开始将正式进入本游戏的开发过程，首先开发的是游戏的主控制类BasketBall_Shot_Activity类，该类的主要作用是在适当的时间初始化相应的界面，并根据其他界面发送来的消息切换到用户需要的界面。开发该类的详细步骤如下。

（1）首先开发的是BasketBall_Shot_Activity类的代码框架，其框架的详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的BasketBall_Shot_Activity.java。

```
package com.bn.tl;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘的源代码
enum WhichView {WELCOME_VIEW,CAIDAN_VIEW,SHEZHI_VIEW,//相关枚举
    GUANYU_VIEW,GAME_VIEW,BANGZHU_VIEW,JILU_VIEW,OVER_VIEW,JIAZAI_VIEW}
public class BasketBall_Shot_Activity extends Activity {
    WhichView curr;                                   //当前枚举值
    ……//此处省略相关成员变量的声明代码，读者可自行查阅随书光盘的源代码
    AndroidVersionDialog avDialog;                    //对话框
   @Override
       public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
            chushihuaSounds();                            //初始化声音池
            chushihuaScreen();                            //初始化屏幕的分辨率
            curr=WhichView.WELCOME_VIEW;                  //标记当前界面为菜单界面
            new Thread(){                                 //创建一个线程来创建需要的资源
            ……//此处省略了部分代码，将在后面的步骤中给出
            }.start();
          xiaoxichuli=new Handler(){                      //初始化消息处理器
            ……//此处省略了部分代码，将在后面的步骤中给出
            };
            xiaoxichuli.sendEmptyMessage(CAIDAN_RETRY);//这里跳到菜单界面
          if(Build.VERSION.SDK_INT<Build.VERSION_CODES.FROYO) {//查看当前Android版本
          this.showDialog(2);                 //如果版本低于2.2则弹出对话框
          }
          else if(this.getGLVersion()<2) {  //判断当前系统所支持的最高OpenGLES版本是不是大于2
```

```
26            this.showDialog(1);                          //如果不是，则弹出对应对话框
27        }}
28        public int getGLVersion(){                       //获取 OpenGLES 所支持的最高版本
29        ……//此处省略了部分代码，将在后面的步骤中给出
30        }
31        public Dialog onCreateDialog(int id)  {          //返回对应的 Dialog
32         ……//关于对话框的内容不是本书重点，对于该方法本节将不再进行介绍
33        }
34        public void onPrepareDialog(int id, Dialog dialog) {
                                                           //对对应的对话框中的按钮作相应处理
35         ……//关于对话框的内容不是本书重点，对于该方法本节将不再进行介绍
36        }
37        public void chushihuaScreen(){                   //初始化屏幕分辨率
38        ……//此处省略了部分代码，将在后面的步骤中给出
39        }
40        @Override
41         public boolean onKeyDown(int keyCode, KeyEvent e) { //键盘监听方法
42         ……//该方法的内容不是本书介绍的重点，对于该方法本节将不再进行介绍
43        }
44        public void chushihuaSounds(){                          //创建声音的方法
45        ……//该方法的内容不是本书介绍的重点，对于该方法本节将不再进行介绍
46        }
47        public void shengyinBoFang(int sound,int loop) {      //播放声音的方法
48         ……//该方法的内容不是本书介绍的重点，对于该方法本节将不再进行介绍
49    }}
```

- 第10～21行为重写的onCreate方法，当运行该类时，首先调用此方法，在此方法中首先进行声音和屏幕分辨率的初始化，然后在代码第15～17行新建一个线程来对加载界面下需要加载的资源进行初始化，然后跳转到主菜单界面。
- 第22～27行为判断该手机的系统版本是否为2.2以上和其支持的OpenGL ES版本是否为2.0，如果不是则弹出对应的对话框。第28～30行为获取该手机对OpenGL ES所支持的最高版本。
- 第31～36行为对话框的相应处理，其中31～33行为根据对应ID生成对应的Dialog对象，第34～36行为对弹出的对话框进行相应的处理。
- 第37～39行为初始化屏幕分辨率，首先获取屏幕分辨率，然后根据480x800的标准来计算得到图片和屏幕坐标的缩放比例，以实现自适应屏幕的效果。
- 第40～49行键盘监听方法、创建声音方法和播放声音方法，其具体代码在后面的步骤中给出。

（2）接下来开发的是BasketBall_Shot_Activity类中，加载系统资源时用到的线程，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的BasketBall_Shot_Activity.java。

```
new Thread(){
        public void run(){
            try{                                       //加载3D中加载界面的纹理资源
        Constant.loadWelcomeBitmap(BasketBall_Shot_Activity.this.getResources(),
                    new int[]{R.drawable.welcome,
        R.drawable.dott,R.drawable.bangzuwelcome}); //加载3D中加载界面的shader字符串
        ShaderManager.loadCodeFromFile(BasketBall_Shot_Activity.this.getResources());
                    SQLiteUtil.initDatabase();     //创建数据库
                }
                    catch(Exception e) {
                        e.printStackTrace();
            }}}.start();                               //启动线程
```

说明

该线程主要是将加载界面下的两幅图片初始化，并加载此界面下的shader字符串，其作用是在单击开始游戏后，切换到3D界面，此时在没有加载完其他资源的时候，只绘制加载界面需要的物体，所以在一开始对其初始化，在切换至3D界面时只是一闪而过，不黑屏。最后创建记录成绩的数据库。

（3）接下来开发的是BasketBall_Shot_Activity中的Handler消息处理器，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的BasketBall_Shot_Activity.java。

```
xiaoxichuli=new Handler(){                              //Handler 消息处理器
        @Override
        public void handleMessage(Message msg) {        //重写方法
            super.handleMessage(msg);                   //调用父类处理
            switch(msg.what) {
            case SHENGYING_KG_JIEMIAN:                  //进入声音设置界面
                curr=WhichView.SHEZHI_VIEW;             //标记当前所在界面为设置界面
                gamesound=new ShengyinKGJiemian(BasketBall_Shot_Activity.this);
                setContentView(gamesound);              //切换到设置界面
                    break;
                case CAIDAN_JIEMIAN:                    //进入菜单选择界面
                    curr=WhichView.CAIDAN_VIEW;         //标记当前所在界面为菜单界面
                caidanjiemian=new CaiDanView(BasketBall_Shot_Activity.this);
                                                        //创建对象
                    setContentView(caidanjiemian); //切换到当前界面为菜单界面
                    break;
                case JIAZAI_JIEMIAN:                        //进入加载主场景中的资源
                    curr=WhichView.JIAZAI_VIEW;             //标记当前所在界面为加载界面
                    gameplay = new GLGameView(BasketBall_Shot_Activity.this);
                    Resources r=BasketBall_Shot_Activity.this.getResources();
                    gameplay.initObjectWelcome(r); //初始化加载界面的相应资源
                    if(isBJmiusic) {                        //如果开启了播放背景音乐
                    beijingyinyue.start();                  //播放背景音乐
                    }
                xiaoxichuli.sendEmptyMessage(YOUXI_JIEMIAN);//发送进入游戏界面的消息
                    break;
                case YOUXI_JIEMIAN:                         //游戏界面
                    flag=true;                              //物理模仿线程标志位
                    isnoPlay=true;                          //是否播放视频
                setContentView(gameplay);                   //这里切换到游戏界面中
                gameplay.requestFocus();                    //获取焦点
```

```
31                  gameplay.setFocusableInTouchMode(true);//设置为可触控
32                  break;
33              case GUANYU_JIEMIAN:                //关于界面
34                  curr=WhichView.GUANYU_VIEW;     //标记当前所在界面为关于界面
35              guanyujiemian=new GuanYuView(BasketBall_Shot_Activity.this);
                                                    //创建对象
36                  setContentView(guanyujiemian); //切换到关于界面
37                  break;
38              case BANGZHU_JIEMIAN:               //帮助界面
39                  curr=WhichView.BANGZHU_VIEW;    //标记当前所在界面为帮助界面
40                  isnoHelpView=true;              //该界面为帮助界面
41                  xiaoxichuli.sendEmptyMessage(JIAZAI_JIEMIAN);//进入加载界面
42                  break;
43              case JIESHU_JIEMIAN:                //游戏结束
44                  curr=WhichView.OVER_VIEW;       //标记当前所在界面为游戏结束界面
45          jieshujiemian=new YouXiuJieShuView(BasketBall_Shot_Activity.this,
            caidanjiemian);
46                  setContentView(jieshujiemian); //切换到游戏结束界面
47                  break;
48              case CAIDAN_RETRY:                  //菜单界面
49                  curr=WhichView.CAIDAN_VIEW;     //标记当前所在界面为菜单界面
50                caidanjiemian=new CaiDanView(BasketBall_Shot_Activity.this);
                                                    //创建对象
51                  setContentView(caidanjiemian); //切换到菜单界面
52                  break;
53              case JILU_JIEMIAN:                  //记录界面
54                  curr=WhichView.JILU_VIEW;       //标记当前所在界面为记录界面
55                  lishijilu=new JiLuView(BasketBall_Shot_Activity.this);
                                                    //创建记录界面
56                  setContentView(lishijilu);      //切换到记录界面
57                  break;
58          }}};
```

- 第6-15行为当发送来的消息为进入设置界面或者菜单界面时，首先标记当前所在界面为对应界面，然后创建对应界面的对象，最后使用setContentView方法切换到对应界面。
- 第16-25行为当发送来的消息为加载界面时，首先标记当前所在界面为加载界面，然后创建游戏界面对象，再创建加载界面需要的资源，如需开启背景音乐则开启背景音乐，最后发送切换到游戏界面的消息。
- 第26-32行为切换到游戏界面；第33-37行为切换到关于界面；第38-42行为切换到帮助界面，切换到帮组界面，只是将游戏界面标志位设为true即可，其和游戏界面没有本质区别；第43-47行为切换到游戏界面；第48-52行为切换到菜单界面；第53-57行为切换到排行榜界面。

（4）接下来开发的是在BasketBall_Shot_Activity中的获取OpenGL ES所支持的最高版本方法。其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的BasketBall_Shot_Activity.java。

```
public int getGLVersion(){                                  //获取 OpenGL ES 所支持的最高版本
    ActivityManager am = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
    ConfigurationInfo info = am.getDeviceConfigurationInfo();
                                                            //从 activity 管理者中得到该对象
    int majorVersion=info.reqGlEsVersion;                   //得到其支持的版本号
    majorVersion=majorVersion>>>16;                         //无符号右移 16 位
    return majorVersion;                                    //返回其支持的最高版本号
}
```

> **说明**
>
> 该方法的作用主要为获取使用设备所支持的OpenGL ES最高版本，从而确定该游戏是否可以运行。

（5）接下来开发的是在BasketBall_Shot_Activity中生成对应Dialog对象的方法，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的BasketBall_Shot_Activity.java。

```
public Dialog onCreateDialog(int id) {                      //创建 Dialog 对象
    Dialog result=null;                                     //声明引用
    switch(id) {
    case 1:                                                 //如果是编号为 1 的对话框
        cvDialog=new CheckVersionDialog(this);              //创建 OpenGL 版本支持不够对话框
        result=cvDialog;
        break;
    case 2:
        avDialog=new AndroidVersionDialog(this);            //创建 Android 系统版本支持不够对话框
            result=avDialog;
            break;
        }
            return result;
      }
```

说明

该方法的作用主要为根据传入不同对话框的id，创建不同对话框对象。

（6）接下来开发的是在BasketBall_Shot_Activity中对对话框进行相应处理的方法，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/t1目录下的BasketBall_Shot_Activity.java。

```java
public void onPrepareDialog(int id, Dialog dialog) {//对话框处理方法
    switch(id){
      case 1:                                    //支持 OpenGL 版本不够对话框
           Button bok=(Button)cvDialog.findViewById(R.id.ok_button);//获得确定按钮的引用
           bok.setOnClickListener(                //对其注册监听
                 new OnClickListener(){
                        @Override
                        public void onClick(View v) {
                             System.exit(0);       //直接退出
                        }});
           break;
           case 2:                                     //支持 Android 系统版本不够
               Button ok=(Button)avDialog.findViewById(R.id.ok);//获得确定按钮的引用
               ok.setOnClickListener(                  //对其添加监听
                     new OnClickListener(){
                            @Override
                            public void onClick(View v) {
                                 System.exit(0);       //直接退出
                            }});
               break;
         }}
```

说明

该方法为对话框创建时动态调用的方法，该方法中给定了当Android系统版本不够和支持OpenGL ES版本不够两种情况下，弹出对话框中按钮的监听事件。

（7）接下来开发的是在BasketBall_Shot_Activity中初始化屏幕分辨率方法，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的BasketBall_Shot_Activity.java。

```java
public void chushihuaScreen(){                                          //初始化屏幕分辨率
    requestWindowFeature(Window.FEATURE_NO_TITLE);                      //去掉通知栏
    getWindow().setFlags (                                              //全屏显示
        WindowManager.LayoutParams.FLAG_FULLSCREEN,
        WindowManager.LayoutParams.FLAG_FULLSCREEN);
        setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);//竖屏显示
      DisplayMetrics dm=new DisplayMetrics();                           //获取屏幕分辨率
      getWindowManager().getDefaultDisplay().getMetrics(dm);
      int tempHeight=(int) (SCREEN_HEIGHT=dm.heightPixels);  //得到屏幕高度
          int tempWidth=(int) (SCREEN_WIDHT=dm.widthPixels);//得到屏幕宽度
          if(tempHeight>tempWidth) {                        //如果此时高度大于宽度
            SCREEN_HEIGHT=tempHeight;                        //则高度赋值为得到的高度
            SCREEN_WIDHT=tempWidth;                          //宽度赋值为得到的高度
          }else{
            SCREEN_HEIGHT=tempWidth;                         //否则高度等于得到的宽度
            SCREEN_WIDHT=tempHeight;                         //宽度等于得到的高度
          }
          float zoomx=SCREEN_WIDHT/480;                      //计算对应 x 轴方向的缩放比
            float zoomy=SCREEN_HEIGHT/800;                   //计算对应 y 轴方向的缩放比
            if(zoomx>zoomy){                                 //如果 x 方向的缩放比大于 y 轴上的
                ratio_width=ratio_height=zoomy;              //则缩放比按照 y 轴方向的缩放比缩放
            }else    {
                ratio_width=ratio_height=zoomx;              //否则按照 x 轴方向的缩放比缩放
            }
            sXtart=(SCREEN_WIDHT-480*ratio_width)/2;//计算 2D 界面绘制图形时的起始位置
            sYtart=(SCREEN_HEIGHT-800*ratio_height)/2;
      }
```

说明

该方法为获取该程序所在手机的屏幕分辨率，然后以分辨率为480×800的手机作为参考，对在其他屏幕分辨率下，对程序中使用的图片素材尺寸进行对应的缩放，使整个程序自适应屏幕分辨率不同的手机。

（8）接下来开发的是在BasketBall_Shot_Activity中的键盘监听方法，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的BasketBall_Shot_Activity.java。

```
1 @Override
2     public boolean onKeyDown(int keyCode, KeyEvent e) {  //键盘监听方法
3     if(keyCode!=4) {                                    //如果不是返回键则不作处理
4         return false;
5     }
6     if(curr==WhichView.SHEZHI_VIEW||curr==WhichView.GUANYU_VIEW
7         ||curr==WhichView.JILU_VIEW||curr==WhichView.BANGZHU_VIEW||
8         curr==WhichView.OVER_VIEW){                     //如果在这几个界面
9         xiaoxichuli.sendEmptyMessage(CAIDAN_RETRY);     //返回到菜单界面
10            return true;
11        }
12        if(curr==WhichView.CAIDAN_VIEW){                //菜单界面
13            System.exit(0);
14            return true;
15        }
16        if(curr==WhichView.GAME_VIEW) {                 //游戏界面
17            isnoHelpView=false;
18            flag=false;                                 //现场停止
19            shipingJs=0;                                //视频现场归零
20            xiaoxichuli.sendEmptyMessage(CAIDAN_RETRY); //返回到菜单界面
21            return true;
22        }
23        return true;
24      }
```

- 第6～11行为当前所在界面为设置界面、关于界面、记录界面、帮助界面或者游戏结束界面时，均返回到菜单界面。
- 第12～15行为该界面在菜单界面时，直接退出；第16～22行为当前所在界面为游戏界面时，首先将物理模拟线程停止，然后返回到菜单界面。

（9）接下来开发的是在BasketBall_Shot_Activity中初始化声音方法，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的BasketBall_Shot_Activity.java。

```
public void chushihuaSounds(){                                          //创建声音的方法
    beijingyinyue=MediaPlayer.create(this,R.raw.beijingyingyu);  //创建背景音乐
    beijingyinyue.setLooping(true);                                    //设置循环播放背景音乐
    beijingyinyue.setVolume(0.2f, 0.2f);                               //设置其声音大小
    shengyinChi=new SoundPool (4, AudioManager.STREAM_MUSIC, 100);//创建声音池
    soundIdMap=new HashMap<Integer,Integer>();               //创建存放声音 Id 的 Map
    soundIdMap.put(1, shengyinChi.load(this,R.raw.pengzhuang,1));//碰撞地板和墙等的声音
```

```
    soundIdMap.put(2, shengyinChi.load(this,R.raw.levelend,1));  //游戏时间结束时的声音
    soundIdMap.put(3, shengyinChi.load(this,R.raw.shoot,1));     //球进网时的声音
    }
```

说明

该方法的作用为采用MediaPlayer创建背景音乐播放器，采用声音池播放碰撞地板和墙面的声音、游戏结束声音以及球进网的声音。

（10）最后开发的是在BasketBall_Shot_Activity中播放声音的方法，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的BasketBall_Shot_Activity.java。

```
public void shengyinBoFang(int sound,int loop) {          //播放声音的方法
    if(!isCJmiusic){                                        //如果为不播放场景音乐则直接返回
        return;
    }
    AudioManager mgr=(AudioManager)this.getSystemService(Context.AUDIO_SERVICE);
    float streamVolumeCurrent=mgr.getStreamVolume(AudioManager.STREAM_MUSIC);
    float streamVolumeMax=mgr.getStreamMaxVolume(AudioManager.STREAM_MUSIC);
    float volume=streamVolumeCurrent/streamVolumeMax;
       shengyinChi.play (
            soundIdMap.get(sound),                         //声音资源 id
            volume,                                        //左声道音量
            volume,                                        //右声道音量
            1,                                             //优先级
            loop,                                          //循环次数 -1 代表永远循环
            0.5f );}                                       //回放速度 0.5f－2.0f
```

说明

该方法为场景音乐播放方法，如果设置为开启场景音乐，则每次调用该方法播放对应的声音时播放声音，否则直接返回。而播放声音时，需要将声音的id、其左右声道音量和循环次数等参数传入。

21.5 辅助界面相关类

前一小节介绍了主控制类BasketBall_Shot_Activit，本小节将对该游戏的辅助界面相关类进行介绍，该游戏的辅助界面主要是主菜单界面CaiDanView类、设置界面ShengyinKGJiemian类、关于界面GuanYuView类和记录界面JiLuView类，下面就对这些类的开发进行详细的介绍。

21.5.1 主菜单界面CaiDanView类

本节将对该游戏的第一个界面即主菜单界面进行介绍，该界面是游戏启动后，第一个出现的界面，在该界面下可选择对应的按钮进入对应的功能界面，开发该类的详细步骤如下。

（1）首先开发的是主菜单界面的框架，CaiDanView类的代码框架如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的CaiDanView.java。

```
package com.bn.tl;
```

```
2 ……//此处省略了部分类和包的引入代码，读者可自行查阅光盘的源代码
3 public class CaiDanView extends SurfaceView implements SurfaceHolder.Callback{
4     BasketBall_Shot_Activity activity;                          //Activity 的引用
5     ……//此处省略部分成员变量的声明代码，读者可自行查阅随书光盘的源代码 //是否按下退出按钮处
6     float left1=LEFT*ratio_width+sXtart;                        //按钮的起始位置
7     public CaiDanView(BasketBall_Shot_Activity activity) {      //构造器
8         super(activity);                                        //调用父类处理
9         this.activity=activity;                                 //activity 引用
10            this.getHolder().addCallback(this);     //设置生命周期回调接口的实现者
11            paint = new Paint();                    //创建画笔
12            paint.setAntiAlias(true);               //打开抗锯齿
13            initBitmap();                           //初始化图片
14        }
15        public void onDraw(Canvas canvas)  {
16        ……//此处省略了详细绘制方法中的部分代码，将在后面的步骤中给出
17        }
18        @Override
19        public boolean onTouchEvent(MotionEvent e){
20        ……//此处省略了屏幕监听的部分代码，将在后面的步骤中给出
21        }
22        public void doDraw()  {
23        ……//此处省略绘制方法中的部分代码，将在后面的步骤中给出
24        }
25        @Override
26        public void surfaceChanged(SurfaceHolder holder, int format, int width,int height) {}
27        @Override
28        public void surfaceCreated(SurfaceHolder holder) {
29            this.holder=holder;                         //得到 SurfaceHolder 对象
30            doDraw();                                   //调用绘制方法进行绘制
31        }
32        @Override
33        public void surfaceDestroyed(SurfaceHolder holder) {}
34        public void initBitmap(){                       //初始化图片方法
35        ……//此处省略了初始化图片方法的部分代码，将在后面的步骤中给出
36        }}
```

- 第15-17行为对整个菜单界面的具体绘制方法，其就是将背景图片和各个按钮图片绘制到界面上，当玩家触摸到按钮并且未抬起的情况下会对该按钮进行放大绘制。
- 第18-21行为屏幕监听方法，该方法根据玩家触摸屏幕的情况来进行对应的操作。

（2）接下来开发该类中的onDraw方法，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的CaiDanView.java。

```
public void onDraw(Canvas canvas)  {                           //界面绘制方法
        float xstar=25*ratio_width;                            //计算整个界面的起始位置
        float ystar=6*ratio_height;
        if(canvas==null) return;                               //如果画布为空则返回
        super.onDraw(canvas);
        canvas.drawBitmap(beijingtupian, sXtart, sYtart, paint); //绘制背景
        if(isKaishi){                                          //如果触摸到开始游戏按钮未抬起

            canvas.drawBitmap(scaleToFit(kaishi,1.2f,1.2f), left1-xstar,
                    160*ratio_height+sYtart-ystar, paint); //放大绘制开始按钮
            }else{
            canvas.drawBitmap(kaishi, left1, 160*ratio_height+sYtart, paint);
                                                            //否则按照原比例绘制
            }
            if(isshezhi){                                   //如果触摸到设置按钮未抬起
                canvas.drawBitmap(scaleToFit(shezhi,1.2f,1.2f), left4-xstar,
                        240*ratio_height+sYtart-ystar, paint); //放大绘制设置按钮
            }else    {
                canvas.drawBitmap(shezhi, left4,240*ratio_height+sYtart, paint);
                                                            //否则原比例绘制
            }
            if(isguanyu){                                   //如果触摸到关于按钮未抬起
                canvas.drawBitmap(scaleToFit(guanyu,1.2f,1.2f), left3-xstar,
                        320*ratio_height+sYtart-ystar, paint); //放大绘制关于按钮
            }else{
            canvas.drawBitmap(guanyu, left3, 320*ratio_height+sYtart, paint);
                                                            //否则按原比例绘制
            }
            if(isbangzhu) {                                 //如果触摸到帮助按钮未抬起
                canvas.drawBitmap(scaleToFit(bangzhu,1.2f,1.2f), left2-xstar,
                        400*ratio_height+sYtart-ystar, paint); //放大绘制帮助按钮
            }else{
            canvas.drawBitmap(bangzhu, left2, 400*ratio_height+sYtart, paint);
                                                            //否则按原比例绘制
            }
            if(islishijilu){                                //如果触摸到记录按钮未抬起
                canvas.drawBitmap(scaleToFit(lishijilu,1.2f,1.2f), left4-xstar,
                        480*ratio_height+sYtart-ystar, paint); //放大绘制记录按钮
            }else{
            canvas.drawBitmap(lishijilu, left4, 480*ratio_height+sYtart, paint);
                                                            //否则按原比例绘制
            }
            if(istuichu){                                   //如果触摸到退出按钮未抬起
                canvas.drawBitmap(scaleToFit(tuichu,1.2f,1.2f), left4-xstar,
                        560*ratio_height+sYtart-ystar, paint); //放大绘制退出按钮
            }else{
                canvas.drawBitmap(tuichu, left4, 560*ratio_height+sYtart, paint);
                                                            //否则按原比例绘制
            }}
```

说明

该方法为具体绘制方法，将背景和各个按钮绘制到画布上，当玩家触摸屏幕，触摸到某个按钮未抬起时将该按钮放大绘制，只有

在按下和抬起时都在此按钮上时才算按下了此按钮。

（3）接下来开发的是该类中对屏幕监听处理的onTouchEvent方法，其详细代码如下。

代码位置：见随书光盘中源代码/第21

章/BasketBall_Shot/src/com/bn/tl目录下的CaiDanView.java。

```
@Override
    public boolean onTouchEvent(MotionEvent e){        //屏幕监听方法
        float x=e.getX();                               //获取触摸点的 x、y 坐标
        float y=e.getY();
        switch(e.getAction()) {
        case MotionEvent.ACTION_DOWN:                   //如果是按下屏幕
            if(x>left1&&x<left1+250*ratio_width&&y>160*ratio_height+sYtart&&
            y<160*ratio_height+sYtart+60*ratio_height){ //触摸点在开始游戏按钮范围内
                isKaishi=true;                          //将触摸了开始按钮的标志位置为 true
                }else{
                    isKaishi=false;                     //否则置为 false
                }
                if(x>left1&&x<left1+250*ratio_width&&y>240*ratio_height+sYtart&&
                    y<240*ratio_height+sYtart+60*ratio_height){ //触摸点在设置按钮范围内
                    isshezhi=true;                      //将触摸了设置按钮的标志位置为 true
                }else{
                    isshezhi=false;                     //否则置为 false
                }
                if(x>left1&&x<left1+250*ratio_width&&y>320*ratio_height+sYtart&&
                    y<320*ratio_height+sYtart+60*ratio_height){ //触摸点在关于按钮范围内
                    isguanyu=true;                      ///将触摸了关于按钮的标志位置为 true
                }else{
                    isguanyu=false;                     //否则置为 false
                }
                if(x>left1&&x<left1+250*ratio_width&&y>400*ratio_height+sYtart&&
                    y<400*ratio_height+sYtart+60*ratio_height){ //触摸点在帮助按钮范围内
                    isbangzhu=true;                     //将触摸了帮助按钮的标志位置为 true
                }else{
                    isbangzhu=false;                    //否则置为 false
                }
                if(x>left1&&x<left1+250*ratio_width&&y>480*ratio_height+sYtart&&
                    y<480*ratio_height+sYtart+60*ratio_height){ //触摸点在记录按钮范围内
                    islishijilu=true;                   //将触摸了记录按钮的标志位置为 true
                }else{
                    islishijilu=false;                  //否则置为 false
                }
                if(x>left1&&x<left1+250*ratio_width&&y>560*ratio_height+sYtart&&
                    y<560*ratio_height+sYtart+60*ratio_height){ //触摸点在退出按钮范围内
                    istuichu=true;                      //将触摸了退出按钮的标志位置为 true
                }else{
                    istuichu=false;                     //否则置为 false
                }
                 doDraw();                              //调用方法重绘
                break;
            case MotionEvent.ACTION_UP:                 //如果是触摸时抬起的动作
                if(isKaishi&&x>left1&&x<left1+250*ratio_width&&y>160*ratio_height
                +sYtart&&
            y<160*ratio_height+sYtart+60*ratio_height){ //按下点和抬起点都在开始按钮的范围
                    isKaishi=false;                     //对应标志位置为 false
                    isnoHelpView=false;                 //是否为帮助界面设为 false
                    activity.xiaoxichuli.sendEmptyMessage(JIAZAI_JIEMIAN);
```

```
                                                    //切换到加载界面
51                  }else{
52                      isKaishi=false;                 //否则将其对应标志位置为 false
53                  }
54                  if(isshezhi&&x>left1&&x<left1+250*ratio_width&&y>240*ratio_height
                    +sYtart&&
55              y<240*ratio_height+sYtart+60*ratio_height){ //按下点和抬起点都在设置按钮的范围
56                      isshezhi=false;                 //对应标志位置为 false
57              activity.xiaoxichuli.sendEmptyMessage(SHENGYING_KG_JIEMIAN); //切换设置界面
58                  }else{
59                      isshezhi=false;                 //否则将其对应标志位置为 false
60                  }
61                  if(isguanyu&&x>left1&&x<left1+250*ratio_width&&y>320*ratio_height
                    +sYtart&&
62              y<320*ratio_height+sYtart+60*ratio_height){//按下点和抬起点都在关于按钮的范围
63                      isguanyu=false;                 //对应标志位置为 false
64                  activity.xiaoxichuli.sendEmptyMessage(GUANYU_JIEMIAN);//切换到关于界面
65                  }else{
66                      isguanyu=false;                 //否则将其对应标志位置为 false
67                  }
68                  if(isbangzhu&&x>left1&&x<left1+250*ratio_width&&y>400*ratio_height
                    +sYtart&&
69              y<400*ratio_height+sYtart+60*ratio_height){ //按下点和抬起点都在帮助按钮的范围
70                      isbangzhu=false;                //对应标志位置为 false
71              activity.xiaoxichuli.sendEmptyMessage(BANGZHU_JIEMIAN);//切换到帮助界面
72                  }else{
73                      isbangzhu=false;                //否则将其对应标志位置为 false
74                  }
76                  if(islishijilu&&x>left1&&x<left1+250*ratio_width&&y>480*ratio_
                    height+sYtart&&
77              y<480*ratio_height+sYtart+60*ratio_height){ //按下点和抬起点都在记录按钮的范围
78                      islishijilu=false;              //对应标志位置为 false
79                      activity.xiaoxichuli.sendEmptyMessage(JILU_JIEMIAN);
80                  }else{
81                      islishijilu=false;              //否则将其对应标志位置为 false
82                  }
83                  if(istuichu&&x>left1&&x<left1+250*ratio_width&&y>560*ratio_height
                    +sYtart&&
84              y<560*ratio_height+sYtart+60*ratio_height){ //按下点和抬起点都在退出按钮的范围
85                      istuichu=false;                 //对应标志位置为 false
86                      System.exit(0);                 //直接退出
87                  }else{
88                      istuichu=false;                 //否则将其对应标志位置设为 false
89                  }
90                  doDraw();                           //重绘
91              }
92              return true;
93          }
```

- 第6-44行为触摸按下时的处理，根据触摸点的位置判断其是否触摸到某个按钮，如果触摸到某个按钮了，则将该按钮放大绘制。
- 第45-90行为触摸抬起时的处理，同样根据抬起点的坐标，如果按下和抬起都在同一点且在某个按钮上，则执行对应的动作，如按下和抬起均在开始游戏按钮的范围内，则发送切换到加载界面的消息来切换到加载界面。

（4）接下来开发的是该类中重绘方法doDraw，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/t1目录下的CaiDanView.java。

```
public void doDraw()  {
        canvas=holder.lockCanvas();                          //得到Canvas对象
        try{
            synchronized(holder){                            //锁定画布
                onDraw(canvas);                              //调用具体绘制方法进行重绘
            }
        }catch(Exception e)   {                              //异常处理
            e.printStackTrace();
        }finally{
                if(canvas!=null)  {
                    holder.unlockCanvasAndPost(canvas);//释放画布
                }}}
```

说明

该方法的作用主要为锁定二维游戏画布，进行平面贴图，当绘制完毕后释放画布，该方法执行结束。

（5）最后开发的是在该类中的图片初始化方法initBitmap，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/t1目录下的CaiDanView.java。

```
public void initBitmap(){
        beijingtupian=scaleToFit(BitmapFactory.decodeResource(getResources(),
            R.drawable.background),ratio_width,ratio_height);    //背景图片
        kaishi=scaleToFit(BitmapFactory.decodeResource(getResources(),
            R.drawable.begin),ratio_width,ratio_height);         //开始游戏按钮
        tuichu=scaleToFit(BitmapFactory.decodeResource(getResources(),
            R.drawable.shut),ratio_width,ratio_height);          //退出按钮
        guanyu=scaleToFit(BitmapFactory.decodeResource(getResources(),
            R.drawable.about1),ratio_width,ratio_height);        //关于按钮
            bangzhu=scaleToFit(BitmapFactory.decodeResource(getResources(),
                R.drawable.help1),ratio_width,ratio_height);     //帮助按钮
            lishijilu=scaleToFit(BitmapFactory.decodeResource(getResources(),
                R.drawable.jilu),ratio_width,ratio_height);      //记录按钮
            shezhi=scaleToFit(BitmapFactory.decodeResource(getResources(),
                R.drawable.shezhi),ratio_width,ratio_height);    //设置按钮
        }
```

说明

该方法为初始化图片方法，首先将图片加载进来，然后根据屏幕初始化的结果，将其按照需要的比例缩放，也达到自适应屏幕的效果。

21.5.2 设置界面ShengyinKGJiemian类

前面对菜单界面的开发进行了介绍，在菜单界面下单击设置按钮后就进入了设置界面，下面就 对设置界面ShengyinKGJiemian类进行介绍，具体的开发步骤如下。

（1）首先开发的是ShengyinKGJiemian类的代码框架，具体代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的ShengyinKGJiemian.java。

```
package com.bn.tl;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘的源代码
public class ShengyinKGJiemian extends SurfaceView implements SurfaceHolder.Callback{
    BasketBall_Shot_Activity activity;                  //Activity 的引用
    ……//此处省略部分成员变量的声明代码，读者可自行查阅随书光盘的源代码
        //是否按下时在返回按钮上
    boolean isnoQueDing;                           //是否按下时在确定按钮上
    public ShengyinKGJiemian(BasketBall_Shot_Activity activity)  {
        super(activity);
        this.activity=activity;
            this.getHolder().addCallback(this);         //设置生命周期回调接口的实现者
            paint=new Paint();                          //创建画笔
            paint.setAntiAlias(true);                   //打开抗锯齿
            initBitmap();                               //初始化图片
        }
        public void onDraw(Canvas canvas)  {
        ……//此处省略了详细绘制方法中的部分代码，将在后面的步骤中给出
        }
        @Override
        public boolean onTouchEvent(MotionEvent e){
        ……//此处省略了屏幕监听的部分代码，将在后面的步骤中给出
        }
        @Override
        public void surfaceChanged(SurfaceHolder arg0, int arg1, int arg2, int arg3){
        }
        @Override
        public void surfaceCreated(SurfaceHolder holder){
            this.holder=holder;
            onDrawcanvas();
        }
        public void onDrawcanvas(){
        ……//此处省略绘制方法中的部分代码，将在后面的步骤中给出
        }
        @Override
        public void surfaceDestroyed(SurfaceHolder holder){
        }
        public void initBitmap(){
        ……//此处省略了初始化图片方法的部分代码，将在后面的步骤中给出
        }}
```

说明

该框架和主菜单界面类的框架没有本质的区别，此处就不再对其进行详细的介绍，请读者自行查阅随书光盘中的源代码。

（2）接下来开发的是该类中的绘制方法onDraw，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的

ShengyinKGJiemian.java。

```
1 public void onDraw(Canvas canvas)  {
2         super.onDraw(canvas);
3         canvas.drawBitmap(beijing, sXtart, sYtart, paint);  //绘制背景
4         canvas.drawBitmap(isnoBeijing, sXtart+80*ratio_width,
5                     sYtart+300*ratio_height, paint);   //绘制是否播放背景音乐文字图片
6         canvas.drawBitmap(isnoChangjing, sXtart+80*ratio_width,
7                     sYtart+420*ratio_height, paint);   //绘制是否播放场景音乐文字图片
8         canvas.drawBitmap(baiseFangfe, sXtart+350*ratio_width,
9                     sYtart+310*ratio_height, paint);//绘制是否开启背景音乐文字右边方格
10            canvas.drawBitmap(baiseFangfe, sXtart+350*ratio_width,
11                        sYtart+430*ratio_height, paint);
                                                    //绘制是否开启场景音乐文字右边方格
12            if(isBJmiusic){                         //如果当前为开启背景音乐,
13            canvas.drawBitmap(honGou, sXtart+350*ratio_width, sYtart+310*ratio_height,
              paint);
14            }                                       //绘制开启背景音乐对应标志钩
15            if(isCJmiusic){                         //如果当前为开启场景音乐
16            canvas.drawBitmap(honGou, sXtart+350*ratio_width, sYtart+430*ratio_height,
              paint);
17            }                                       //绘制开启场景音乐对应标志钩
18            if(isnoFanhui){                         //如果触摸按下时在确定按钮范围
19            canvas.drawBitmap(scaleToFit(fanHui,1.2f,1.2f), //放大绘制确定按钮
20                        sXtart+225*ratio_width, sYtart+572*ratio_height, null);
21            }else{                                  //否则按照原比例绘制
22            canvas.drawBitmap(fanHui, sXtart+250*ratio_width, sYtart+580*ratio_height,
              null);
23            }
24            if(isnoQueDing){                        //如果触摸按下时在返回按钮范围
25            canvas.drawBitmap(scaleToFit(queDing,1.2f,1.2f),    //放大绘制返回按钮
26                        sXtart+55*ratio_width, sYtart+572*ratio_height, null);
27            }else{                                  //否则按照原比例绘制
28            canvas.drawBitmap(queDing, sXtart+70*ratio_width, sYtart+580*ratio_height,
              null);
29            }}
```

- 第3-11行为绘制背景、是否开启背景音乐文字图片、是否开启场景音乐文字图片和两个文字对应的方框，该方框中有勾代表为开启，否则为关闭。
- 第12-17行为根据当前设置的状态，如果当前为开启背景音乐，那么在该文字图片后面的方框中绘制勾，否则不绘制，如果当前为开启场景音乐，那么在该文字图片后面的方框中绘制勾，也表示当前设置为开启了场景音乐。
- 第18-29行为绘制确定按钮和返回按钮，当触摸按下在某个按钮上时，将其放大绘制，否则按照原比例绘制。

（3）接下来开发的是该类中的屏幕监听处理方法onTouchEvent，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的ShengyinKGJiemian.java。

```
@Override
    public boolean onTouchEvent(MotionEvent e){
        float x=e.getX();                                   //得到触摸点的位置
        float y=e.getY();
        switch(e.getAction()){
        case MotionEvent.ACTION_DOWN:                       //如果是按下
    if(x>sXtart+50*ratio_width&&x<sXtart+70*ratio_width+180*ratio_width&&
    y>sYtart+580*ratio_height&&y<sYtart+640*ratio_height){//如果按下是在确定按钮范围内
                isnoQueDing=true;                           //对应标志位置为 true，绘制时放大绘制
                }else{
                    isnoQueDing=false;                      //否则对应标志位置为 false
                }
        if(x>sXtart+250*ratio_width&&x<sXtart+250*ratio_width+160*ratio_width&&
        y>sYtart+580*ratio_height&&y<sYtart+640*ratio_height){
                                                            //如果按下是在返回按钮范围内
                    isnoFanhui=true;                        //对应标志位置 true，绘制时放大绘制
                }else{
                    isnoFanhui=false;                       //否则对应标志位置为 false
                }
                onDrawcanvas();                             //重绘
                break;
            case MotionEvent.ACTION_UP:                     //如果当前动作为抬起
        if(isnoQueDing&&x>sXtart+70*ratio_width&&x<sXtart+70*ratio_width+160*ratio_
        width&&
        y>sYtart+580*ratio_height&&y<sYtart+640*ratio_height){
                                                            //按下和抬起都在确定按钮范围内
                    isnoQueDing=false;                      //对应标志位置为 false
                    activity.xiaoxichuli.sendEmptyMessage(CAIDAN_RETRY);
                                                            //返回到主菜单界面
                }else{
                    isnoQueDing=false;                      //否则对应标志位置为 false
                }
        if(isnoFanhui&&x>sXtart+250*ratio_width&&x<sXtart+250*ratio_width+160*
        ratio_width&&
        y>sYtart+580*ratio_height&&y<sYtart+640*ratio_height){
                                                            //按下和抬起均在返回按钮范围内
                    isnoFanhui=false;                       //对应标志位置为 false
                    activity.xiaoxichuli.sendEmptyMessage(CAIDAN_RETRY);//返回到菜单界面
                }else{
                    isnoFanhui=false;                       //否则对应标志位置为 false
                }
        if(x>=sXtart+340*ratio_width&&x<=sXtart+400*ratio_width&&y>=sYtart+300*
        ratio_height&&
                    y<=sYtart+360*ratio_height){    //如果抬起在开启背景音乐右边方框内
                    if(isBJmiusic){                 //如果当前开启了背景音乐
                        activity.beijingyinyue.stop(); //则停止背景音乐播放
                        activity.beijingyinyue=null;   //将其引用指向空引用
                    }else{                          //如果当前为关闭背景音乐
                        if(activity.beijingyinyue==null){  //播放器为空
                    activity.beijingyinyue=MediaPlayer.create(activity,R.raw.
                    beijingyingyu);
                    activity.beijingyinyue.setLooping(true);
                                                    //创建背景音乐并对其进行对应的设置
```

```
45                      activity.beijingyinyue.setVolume(0.2f, 0.2f);//声音大小
46                  }}
47              isBJmiusic=!isBJmiusic;          //是否播放背景音乐，标志置反即可
48          }
49 else if(x>=sXtart+340*ratio_width&&x<=sXtart+400*ratio_width&& y>=sYtart+420*
ratio_height&&
50          y<=sYtart+480*ratio_height){     //抬起处在是否开启场景音乐，右边方框内
51              isCJmiusic=!isCJmiusic;          //是否开启场景音乐，标志位置反
52          }
53          onDrawcanvas();                  //重绘
54      }
55      return true;                         //返回
56  }
```

- 第5-20行为触摸手指按下时的处理代码，该段代码主要是查看其按下时是否在确定或者返回按钮上，如果是将其对应标志位置为true，在绘制时放大绘制对应按钮。
- 第21-53行为触摸手指抬起时的处理代码，该段代码根据抬起时的坐标，首先查看此位置是否在确定按钮、返回按钮或声音设置按钮的区域内。若是则执行相应的处理代码，详细情况请参见上述代码片段中的注释。

> **说明**
>
> 对于在该类中的重绘方法和图片初始化方法中省略的代码段，此处就不再做过多的介绍，其和在主菜单界面类中对应的这两个方法没有本质的区别，请读者自行查阅随书光盘中的源代码。

21.5.3　关于界面GuanYuView类

接下来介绍的是该游戏中的关于界面，该界面主要是对该游戏的一些版本等问题的说明，只是贴了几幅图片，没那么复杂，开发该类的详细步骤如下。

（1）首先开发的是GuanYuView类的框架。其框架代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的GuanYuView.java。

```
1 package com.bn.tl;
2 ……//此处省略了部分类和包的引入代码，读者可自行查阅光盘的源代码
3 public class GuanYuView extends SurfaceView implements SurfaceHolder.Callback{
4     BasketBall_Shot_Activity activity;
5     ……//此处省略了部分类和包的引入代码，读者可自行查阅光盘的源代码
6     boolean isnoQueDing;                                    //是否按下时在确定按钮上
7     public GuanYuView(BasketBall_Shot_Activity activity) {    //构造器
8         super(activity);
9         this.activity=activity;
10            this.getHolder().addCallback(this);          //设置生命周期回调接口的实现者
11            paint=new Paint();                           //创建画笔
12            paint.setAntiAlias(true);                    //打开抗锯齿
13            initBitmap();                                //初始化图片
14            timeStartX=sXtart+15*ratio_width;            //滚动背景的起始位置
15            timeStartY=sYtart+150*ratio_height;
16            timeStartXCaiJian=timeStartX;                      //裁剪框的起始位置
17            timeStartYCaiJian=sYtart+timeStartY+4*ratio_height;
18            caijianWidth=450*ratio_width;
19            caijianHeight=shijianBeijin.getHeight();           //裁剪框的宽高
20            timegradeinitX=timeStartX;
21            timegradeinitY=timeStartY+4*ratio_height;          //滚动方格的初始位置
22            timegradeStartX=timegradeinitX;
23            timeGradeStartY=timegradeinitY;                    //绘制滚动时的起始位置
24        }
25        public void onDraw(Canvas canvas)  {
26        ……//此处省略了详细绘制方法中的部分代码，将在后面的步骤中给出
27        }
28        @Override
29        public boolean onTouchEvent(MotionEvent e){
30        ……//此处省略了屏幕监听的部分代码，将在后面的步骤中给出
31        }
32        @Override
33        public void surfaceChanged(SurfaceHolder arg0, int arg1, int arg2, int arg3){}
34        @Override
35        public void surfaceCreated(SurfaceHolder holder){
36            this.holder=holder;
37            onDrawcanvas();
38        }
39        @Override
40        public void surfaceDestroyed(SurfaceHolder holder){  }
41        public void initBitmap(){
42        ……//此处省略了初始化图片方法的部分代码，将在后面的步骤中给出
43        }
44        public void onDrawcanvas(){
45        ……//此处省略绘制方法中的部分代码，将在后面的步骤中给出
46        }
47    }
```

- 第7-24行为该类的构造器，其中38-47行为计算裁剪框的起始位置，该界面为文字只在中间一块显示，如果图片太长，那么需要拖动查看，所以需要将不必要的部分裁剪出来。
- 在第25-47行中，有部分方法中省略了其代码，其中第25-27行为界面的具体绘制方法，第28-31行为屏幕监听方法，第39-43行为初始化图片方法，第44-47行为重绘方法。

（2）接下来开发的是该类中的绘制方法onDraw，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的GuanYuView.java。

```
public void onDraw(Canvas canvas)  {
        super.onDraw(canvas);
        canvas.drawBitmap(beijing, sXtart, sYtart, null);        //背景
        canvas.drawBitmap(shijianBeijin,timeStartX, timeStartY, null);//方框背景
        canvas.save();                                           //保存当前画面
        canvas.clipRect(new RectF(timegradeinitX,timegradeinitY,//创建剪切矩形
                timeStartX+caijianWidth,timegradeinitY+caijianHeight));
    canvas.drawBitmap(wenzitupian,timegradeStartX, timeGradeStartY, null);//绘制文字图片
    float gundontiaoheight=(-timeGradeStartY+timegradeinitY)*(caijianHeight-50*
    ratio_height)/
            (wenzitupian.getHeight()-caijianHeight)+timegradeinitY;
                                               //计算滚动条y方向的位置
            if(isnoDianJi){                    //如果是触摸滚动则绘制滚动条
   canvas.drawBitmap(gundontiao, timeStartX+caijianWidth-12*ratio_width, gundontiaoheight,
    null);
            }
            canvas.restore();                  //恢复后面的内容不再只绘制在矩形内
            if(isnoQueDing){                             //如果触摸到确定按钮
            canvas.drawBitmap(scaleToFit(queDing,1.2f,1.2f), sXtart+35*ratio_width,
                    sYtart+ 692*ratio_height, null);     //放大绘制确定按钮
            }else{                             //否则按照原比例绘制
            canvas.drawBitmap(queDing, sXtart+50*ratio_width, sYtart+700*ratio_height,
            null);
            }
            if(isnoFanhui){                    //如果触摸到返回按钮
            canvas.drawBitmap(scaleToFit(fanHui,1.2f,1.2f), sXtart+245*ratio_width,
                sYtart+ 692*ratio_height, null);   //放大绘制返回按钮
            }else{                             //否则按照原比例绘制返回按钮
                canvas.drawBitmap(fanHui, sXtart+270*ratio_width, sYtart+700*ratio_
                height, null);
            }}
```

说明

该方法为绘制整个关于界面里面的各个组成物件，和已经介绍过的主菜单界面中绘制方法没有本质区别，此处就不再对其进行过多的介绍。

（3）接下来开发的是该类中的屏幕监听方法onTouchEvent，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的GuanYuView.java。

```
@Override
    public boolean onTouchEvent(MotionEvent e){
        float x=e.getX();                                   //获取触摸处的坐标
        float y=e.getY();
        switch(e.getAction()) {
        case MotionEvent.ACTION_DOWN:                       //如果为按下
            if(x>timeStartX&&x<timeStartX+caijianWidth&&y>timeStartYCaiJian&&
                y<timeStartYCaiJian+caijianHeight){         //单击处为文字图片区域
                mDounX=x;                                   //记录按下处的坐标
                    mDounY=y;
                    mtimeGradeStartY=timeGradeStartY;       //记录此时图片的 Y 方向坐标
                    isnoDianJi=true;                        //标记位置为 true
                }else{
                    isnoDianJi=false;                       //否则标记位置为 false
                }
                if(x>sXtart+50*ratio_width&&x<sXtart+50*ratio_width+160*ratio_
                width&&y>sYtart+
                    700*ratio_height&&y<sYtart+780*ratio_height){ //如果单击处为确定按钮
                    isnoQueDing=true;                       //标志位置为 true
                }else{
                    isnoQueDing=false;                      //否则标志位置为 false
                }
```

```
            if(x>sXtart+270*ratio_width&&x<sXtart+270*ratio_width+160*ratio_
            width&&y>sYtart+
                    700*ratio_height&&y<sYtart+780*ratio_height){//如果单击处为返回按钮
                    isnoFanhui=true;                    //标志位置为 true
                }else{
                    isnoFanhui=false;                   //否则标志位置为 false
                }
                onDrawcanvas();                         //重绘
                break;
            case MotionEvent.ACTION_MOVE:               //触摸事件为移动事件
                float dy=y-mDounY;                      //计算其触摸移动的 y 方向的距离
                if(isnoDianJi==true){                   //如果按下时在文字区域
                if(mtimeGradeStartY+dy<timegradeinitY+caijianHeight-wenzitupian.
                getHeight()){
                    timeGradeStartY=timegradeinitY+caijianHeight-wenzitupian.
                    getHeight();
                    }                                   //计算其是否到边界，到边界了则不滚动
                    else if(mtimeGradeStartY+dy>timegradeinitY){
                        timeGradeStartY=timegradeinitY;
                    }else{
                        timeGradeStartY=mtimeGradeStartY+dy;    //计算其当前位置
                    }
                     onDrawcanvas();                    //重绘
                }
                break;
            case MotionEvent.ACTION_UP:                 //触摸屏幕事件为抬起
            if(isnoQueDing&&x>sXtart+50*ratio_width&&x<sXtart+50*ratio_width+160
            *ratio_width
            &&y>sYtart+700*ratio_height&&y<sYtart+780*ratio_height){
                                                        //抬起处为确定按钮区域
                    isnoQueDing=false;                  //对应标志位置为 false
                    activity.xiaoxichuli.sendEmptyMessage(CAIDAN_RETRY);
                                                        //返回到菜单界面
                }else{
                    isnoQueDing=false;                  //对应标志位置为 false
                }
            if(isnoFanhui&&x>sXtart+270*ratio_width&&x<sXtart+270*ratio_width+160*
            ratio_width
                &&y>sYtart+700*ratio_height&&y<sYtart+780*ratio_height){
                                                        //抬起处为返回按钮
                    isnoFanhui=false;                   //对应标志位置为 false
                    activity.xiaoxichuli.sendEmptyMessage(CAIDAN_RETRY);
                                                        //返回到菜单界面
                }else{
                    isnoFanhui=false;                   //对应标志位置为 false
                }
                if(isnoDianJi){
                    isnoDianJi=false;                   //对应标志位置为 false
                }
                onDrawcanvas();                         //重绘
                break;
        }
        return true;
    }
```

说明

该方法为屏幕触摸事件处理方法，和已经介绍过的菜单界面里面的屏幕触摸事件处理方法没有本质的区别，此处就不再做过多的介绍，详细情况请见注释。而该类中剩下的重绘方法和图片初始化方法在这里也同样不再对其做详细的介绍了，请读者自行查阅随书光盘中的源代码。

21.5.4　记录界面JiLuView类

本小节将对记录界面JiLuView类进行介绍，该类主要将记录在数据库中的成绩取出显示在该界面下，其显示方式有两种，一种是按时间排序，另一种是按成绩排序，开发该类的详细步骤如下。

（1）首先开发的是记录界面的框架，JiLuView类的代码框架如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的JiLuView.java。

```java
package com.bn.tl;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘的源代码
public class JiLuView extends SurfaceView implements SurfaceHolder.Callback{
……//此处省略了相关成员变量的声明，读者可自行查阅随书光盘的源代码
    public JiLuView(BasketBall_Shot_Activity activity){
        super(activity);
        this.activity=activity;
        this.getHolder().addCallback(this);          //设置生命周期回调接口的实现者
        paint=new Paint();                           //创建画笔
            paint.setAntiAlias(true);                //打开抗锯齿
            initBitmap();                            //初始化图片
            String sql_select="select grade,time from paihangbang  order by grade desc
            limit 0,30;";
            vector=SQLiteUtil.query(sql_select);     //从数据库中取出相应的数据
            fanggeGeshu=vector.size();               //得到行数
            if(fanggeGeshu<12){                      //如果总行数小于 12 则让其显示 12 行
            fanggeGeshu=12;
            }
            timeStartX=sXtart+15*ratio_width;        //滚动背景的起始位置
            timeStartY=sYtart+150*ratio_height;
            timeStartXCaiJian=timeStartX+20*ratio_height;    //裁剪框的起始位置
            timeStartYCaiJian=sYtart+timeStartY+100*ratio_height;
            caijianWidth=430*ratio_width;
            caijianHeight=400*ratio_height;                  //裁剪框的宽高
            timegradeinitX=timeStartXCaiJian;
            timegradeinitY=timeStartYCaiJian;//timeStartY+73*ratio_height;
                                                             //滚动方格的初始位置
            timegradeStartX=timegradeinitX;
            timeGradeStartY=timegradeinitY;                  //绘制滚动时的起始位置
        }
        public void onDraw(Canvas canvas)  {
        ……//此处省略了详细绘制方法中的部分代码，将在后面的步骤中给出
```

```
31          }
32          public void drawRectBeijing(Canvas canvas,float startX,float startY){
33          ……//此处省略了绘制颜色矩形背景方法中的部分代码，将在后面的步骤中给出
34          }
35          public void drawScoreStr(Canvas canvas,String s,int width,int height) {
36          ……//此处省略了绘制字符串方法中的部分代码，将在后面的步骤中给出
37          }
38          public void drawRiQi(Canvas canvas,String s,int width,int height) {
39          ……//此处省略了绘制年月日方法中的部分代码，将在后面的步骤中给出
40          }
41          @Override
42          public boolean onTouchEvent(MotionEvent e){
43          ……//此处省略了屏幕触摸事件处理方法中的部分代码，将在后面的步骤中给出
44          }
45          @Override
46          public void surfaceChanged(SurfaceHolder arg0, int arg1, int arg2, int arg3){}
47          @Override
48          public void surfaceCreated(SurfaceHolder holder){
49              this.holder=holder;
50              onDrawcanvas();
51          }
52          @Override
53          public void surfaceDestroyed(SurfaceHolder holder){ }
54          public void initBitmap(){
55          ……//此处省略了图片初始化方法中的部分代码，将在后面的步骤中给出
56          }
57          public void onDrawcanvas(){
58          ……//此处省略了重绘方法中的部分代码，将在后面的步骤中给出
59          }}
```

- 第5-28行为该类的构造器，其中第12～17行为创建查询成绩的SQL语句，然后调用数据库相关类中的静态查询方法得到查询结果，最后查看其结果集中的数据行数，如果少于12行则在绘制颜色矩形时必须绘制12个颜色背景矩形。
- 第29-31行为界面的主要绘制方法，该方法负责绘制整个界面，包括背景图片、数据显示背景、颜色矩形背景、时间数据、成绩数据、确定按钮和返回按钮等。
- 第32-34行为绘制颜色矩形背景，每一行数据对应一个颜色矩形背景。
- 第35-37行为绘制有数字组成的字符串方法。
- 第38-40行为绘制年月日组成的字符串方法，该方法将其年月日切分开，然后绘制数字以及中间的短线，以此来表示年月日，还将时间也切分开，然后绘制时、分和秒，中间绘制顿号图片。
- 第41-59行为屏幕触摸事件处理方法、图片初始化方法和重绘方法等。

（2）接下来开发的是该类中的绘制场景的onDraw方法，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的JiLuView.java。

```
public void onDraw(Canvas canvas){
        super.onDraw(canvas);
        canvas.drawBitmap(beijing, sXtart, sYtart, null);    //绘制背景
        if(isnoGradePaixu){                                  //如果是按照分数排序
    canvas.drawBitmap(shijianBeijin, timeStartX, timeStartY, null);
                                                   //绘制按照分数排序滚动背景
        }else{                                     //否则绘制按照事件排序滚动背景
            canvas.drawBitmap(shijianBeijin2, timeStartX, timeStartY, null);
        }
        canvas.save();                             //保护当前画面
            canvas.clipRect(new RectF(timegradeinitX,timegradeinitY,//创建剪裁框
                    timeStartX+caijianWidth,timegradeinitY+caijianHeight));
            drawRectBeijing(canvas,timegradeStartX,timeGradeStartY);//绘制颜色条
            for(int i=0;i<vector.size();i++){  //循环绘制排行榜的分数和对应时间
                int j=i;
                if(!isnoGradePaixu){               //如果按照成绩排序
                    j=vector.size()-i-1;
                }
                drawRiQi(canvas,vector.get(j).get(1).toString(),     //绘制时间
                    (int)(timegradeStartX+15*ratio_width),
                    (int)(timeGradeStartY+i*geziHeight+5*ratio_height));
                drawScoreStr(canvas,vector.get(i).get(0).toString(),//绘制分数
                    (int)timegradeStartX+(int)(300*ratio_width),
                    (int)(timeGradeStartY+i*geziHeight+5*ratio_height));
        }
        float gundontiaoheight=(-timeGradeStartY+timegradeinitY)*(caijianHeight-50
        *ratio_height)/
            (fanggeGeshu*geziHeight-caijianHeight)+timegradeinitY;//计算滚动条的位置
            if(isnoDianJi){                                       //如果需要绘制滚动条
        canvas.drawBitmap(gundontiao, timeStartX+caijianWidth-8*ratio_width,
        gundontiaoheight, null);
            }                                          //绘制滚动条
            canvas.restore();                          //恢复画面，后面绘制的不再被剪裁
            if(isnoQueDing){                           //如果到了确定按钮
            canvas.drawBitmap(scaleToFit(queDing,1.2f,1.2f), sXtart+35*ratio_width,
                sYtart+ 692*ratio_height, null);   //放大绘制确定按钮
            }else{                                     //否则按照原比例绘制确定按钮
            canvas.drawBitmap(queDing, sXtart+50*ratio_width, sYtart+700*ratio_height,
            null);
            }
            if(isnoFanhui){                            //如果触摸到了返回按钮
            canvas.drawBitmap(scaleToFit(fanHui,1.2f,1.2f), sXtart+245*ratio_width,
                sYtart+ 692*ratio_height, null);   //放大绘制返回按钮
            }else{                                     //否则按照原比例绘制返回按钮
            canvas.drawBitmap(fanHui, sXtart+270*ratio_width, sYtart+700*ratio_height,
            null);
            }}
```

- 第4-8行为绘制其对应的排序背景，如果是按照分数排序则绘制按照分数排序的图片背景，否则绘制按照时间排序的背景图片。
- 第9-30行为绘制裁剪框中的所有内容，如颜色背景矩形、时间、分数和滚动条等。其中第12行为调用对应方法绘制所有的颜色矩形背景，第13-24行为循环绘制从数据库中取出的数据，将每一项记录的时间和分数绘制到对应地方。第25-29行为绘制滚动条。
- 第31-42行为绘制确定按钮和返回按钮。

（3）接下来开发的是该类中的绘制颜色矩形drawRectBeijing方法，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的JiLuView.java。

```
public void drawRectBeijing(Canvas canvas,float startX,float startY){ //传入起始坐标
        for(int i=0;i<fanggeGeshu;i++){                          //循环绘制颜色矩形
            paint.setARGB(color[i%2][0], color[i%2][1], color[i%2][2],color[i%2]
            [3]);
            Rect r=new Rect((int)(startX),(int)(startY+i*geziHeight),//建立对应坐标的矩形
                    (int) (startX+geziWidth),(int) (startY+(1+i)*geziHeight));
            canvas.drawRect(r, paint);                           //绘制矩形
        }}
```

> **说明**
>
> 该方法的作用为绘制记录界面下显示成绩的背景颜色矩形条。

（4）接下来开发的是该类中的绘制数字字符串的drawScoreStr方法，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的JiLuView.java。

```
public void drawScoreStr(Canvas canvas,String s,int width,int height) {
                                                            //绘制数字字符串方法
    String scoreStr=s;
    for(int i=0;i<scoreStr.length();i++){                   //循环绘制数字字符串
        int tempScore=scoreStr.charAt(i)-'0';               //得到该字符是那个数字
  canvas.drawBitmap(iscore[tempScore], width+i*scoreWidth,height, null);
                                                            //根据数字得到对应图片
    }}
```

说明

该方法的作用为根据传入该方法的参数s，确定要绘制的得分数字图片。

（5）接下来开发的是该类中的绘制年、月、日字符串的方法，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的JiLuView.java。

```
public void drawRiQi(Canvas canvas,String s,int width,int height) {//绘制时间字符串
        String ss[]=s.split("-");                         //切割得到年、月、日、小时、分钟和秒
        for(int i=1;i<ss.length;i++){                     //循环绘制该字符串
        if(ss[ss.length-i].length()<2){                   //如果字符串长度小于2则在前面补0
            ss[ss.length-i]="0"+ss[ss.length-i];
        }
        drawScoreStr(canvas,ss[ss.length-i],width+scoreWidth*((ss.length-i-1)*3+0),
        height);
        if(i<3){                                          //如果到绘制时间数字了则要绘制冒号
       canvas.drawBitmap(maohao,width+scoreWidth*((ss.length-i-1)*3-1),height, null);
                                                          //画冒号
            }
            else if(i==4){                                //在月和日之间绘制横线
            canvas.drawBitmap(hengXian,width+scoreWidth*((ss.length-i-1)*3-1),
            height, null);                                //画横线
            }} }
```

说明

该方法将中间由横行隔开的年、月、日、时、分和秒组成的字符串绘制出来，而在年、月、日之间需要用横行隔开，在时、分、秒之间要用冒号隔开。当然所有的数字都是用数字图片代替绘制在界面上的。

（6）最后开发的是该方法中的屏幕触摸事件处理方法onTouchEvent，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的JiLuView.java。

```
@Override
    public boolean onTouchEvent(MotionEvent e){
        float x=e.getX();                                       //得到触摸点的坐标
        float y=e.getY();
        switch(e.getAction()) {
        case MotionEvent.ACTION_DOWN:                           //如果是按下事件
            if(x>timeStartX&&x<timeStartX+caijianWidth/2&&y>timeStartY&&
                y<timeStartY+74*ratio_height){                  //如果单击按照时间排序
            String sql_select="select grade,time from paihangbang   desc limit 0,30;";
                vector=SQLiteUtil.query(sql_select);            //从数据库中取出相应的数据
                    isnoGradePaixu=false;                       //将按照时间排序标志位置为 false
            }else if(x>timeStartX+caijianWidth/2&&x<timeStartX+caijianWidth&&y>
            timeStartY&&
                    y<timeStartY+74*ratio_height){              //如果单击按照成绩排序
                    isnoGradePaixu=true;                        //将按照时间排序标志位置为 true
            String sql_select="select grade,time from paihangbang  order by grade desc
            limit 0,30;";
                vector=SQLiteUtil.query(sql_select);            //从数据库中取出相应的数据
                }
            if(x>timeStartX&&x<timeStartX+caijianWidth&&y>timeStartYCaiJian+74*
            ratio_height&&
                    y<timeStartYCaiJian+caijianHeight+74*ratio_height){
                                                                //单击处是数据显示区域
                    mDounX=x;                                   //记住按下时的坐标
                    mDounY=y;
                    mtimeGradeStartY=timeGradeStartY;           //记住此时裁剪框的位置
                    isnoDianJi=true;                            //对应标志位置为 true
                }else{
                    isnoDianJi=false;                           //否则对应标志位置为 tfalse
                }
                if(x>sXtart+50*ratio_width&&x<sXtart+50*ratio_width+160*ratio_width&&
                    y>sYtart+700*ratio_height&&y<sYtart+780*ratio_height){
                                                                //单击的是确定按钮
                    isnoQueDing=true;                           //对应标志位置为 true
                }else{
                    isnoQueDing=false;                          //否则对应标志位置为 false
                }
                if(x>sXtart+270*ratio_width&&x<sXtart+270*ratio_width+160*ratio_width&&
                    y>sYtart+700*ratio_height&&y<sYtart+780*ratio_height){
                                                                //单击处为返回按钮
                    isnoFanhui=true;                            //对应标志位置为 true
                }else{
```

```
                    isnoFanhui=false;              //否则对应标志位置为 false
                }
                onDrawcanvas();                    //调用重绘方法重绘
                break;
            case MotionEvent.ACTION_MOVE:          //如果是移动事件
                float dy=y-mDounY;                 //计算在 Y 方向的移动距离
                if(isnoDianJi==true){              //如果按下时在数据显示区域
                if(mtimeGradeStartY+dy<timegradeinitY+caijianHeight-fanggeGeshu*
                geziHeight){
                        timeGradeStartY=timegradeinitY+caijianHeight-fanggeGeshu*
                        geziHeight;
                    }                              //如果使其滚动不超过范围，上下不出界
                    else if(mtimeGradeStartY+dy>timegradeinitY){
                        timeGradeStartY=timegradeinitY;
                    }else{
                        timeGradeStartY=mtimeGradeStartY+dy;    //否则正常滚动
                    }
                     onDrawcanvas();               //重绘
                }
                break;
            case MotionEvent.ACTION_UP:            //如果是抬起事件
            if(isnoQueDing&&x>sXtart+50*ratio_width&&x<sXtart+50*ratio_width+160*
            ratio_width
                    &&y>sYtart+700*ratio_height&&y<sYtart+780*ratio_height){
                    isnoQueDing=false;             //对应标志位置为 false
                activity.xiaoxichuli.sendEmptyMessage(CAIDAN_RETRY);//返回到菜单界面
                }else{
                    isnoQueDing=false;             //否则对应标志位置为 false
                }
            if(isnoFanhui&&x>sXtart+270*ratio_width&&x<sXtart+270*ratio_width+160*
            ratio_width
                    &&y>sYtart+700*ratio_height&&y<sYtart+780*ratio_height){
                    isnoFanhui=false;              //对应标志位置为 false
                    activity.xiaoxichuli.sendEmptyMessage(CAIDAN_RETRY);//返回到菜单界面
                }else{
                    isnoFanhui=false;              //对应标志位置为 false
                }
                if(isnoDianJi){                    //按下时在数据显示区域
                    isnoDianJi=false;              //抬起时对应标志位均置为 false
                }
                onDrawcanvas();                    //重绘一下界面
                break;
            }
            return true;
        }
```

说明

该方法和前面介绍过的主菜单中的对应方法没有本质区别，不同的是，该方法中有了使数据可以随着触控点移动而滚动的功能。其原理为将所有数据绘制在裁剪框中，然后随着触摸移动而计算其数据显示的初始位置即可，该类中后面的初始化图片和重绘方法此处也不再对其做详细的介绍，请读者查阅随书光盘中的源代码。

21.5.5 数据库相关类SQLiteUtil

本节将对该游戏中记录玩家得分数据的数据库的开发做相应的介绍，该类只是提供了几个对数据库操作的方法，因此代码的开发并不复杂，具体代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的SQLiteUtil.java。

```
package com.bn.tl;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘的源代码
public class SQLiteUtil {
    static SQLiteDatabase sld;                                    //数据库连接引用
   public static void createOrOpenDatabase() {                   //创建或打开数据库的方法
    try{
         sld=SQLiteDatabase.openDatabase (                        //只能在自己的包下创建数据库
                  "/data/data/com.bn.tl/mydb", null,             //CursorFactory
       SQLiteDatabase.OPEN_READWRITE|SQLiteDatabase.CREATE_IF_NECESSARY);
         }                                                        //如果已经创建则打开连接
         catch(Exception e)    {                                  //异常处理
             e.printStackTrace();
         }}
        public static void closeDatabase(){                       //关闭数据库的方法
         try    {
             sld.close();                                         //关闭数据库
         }catch(Exception e)   {e.printStackTrace();} }           //异常处理
        public static void createTable(String sql) {              //建表方法
         createOrOpenDatabase();                                  //打开数据库
         try{
             sld.execSQL(sql);                                    //建表
         }catch(Exception e){ e.printStackTrace();}
         closeDatabase();                                         //关闭数据库
        }
        public static void insert(String sql) {                   //插入记录的方法
         createOrOpenDatabase();                                  //打开数据库
         try{
             sld.execSQL(sql);                                    //执行 SQL 语句插入一条记录
         }catch(Exception e) { e.printStackTrace();}
             closeDatabase();                                     //关闭数据库
        }
        public static Vector<Vector<String>> query(String sql)  { //查询的方法
         createOrOpenDatabase();                                      //打开数据库
         Vector<Vector<String>> vector=new Vector<Vector<String>>();//新建存放查询结果的向量
         try {
              Cursor cur=sld.rawQuery(sql, new String[]{});  //查询得到结果集
             while(cur.moveToNext())    {                 //循环结果即将结果数据存放到向量中
                  Vector<String> v=new Vector<String>();
                  int col=cur.getColumnCount();        //返回每一行都多少字段
                  for( int i=0;i<col;i++)    {
                          v.add(cur.getString(i));   //将每列的数据存入行向量
                      }
                      vector.add(v);                 //将每一行的向量添加进总向量
```

```
44            }
45            cur.close();                                  //关闭结果集
46        }catch(Exception e) {e.printStackTrace();}
47            closeDatabase();                              //关闭数据库
48            return vector;                                //返回结果数据向量
49      }
50      public  static void initDatabase(){               //创建表
51       String sql="create table if not exists paihangbang(grade int(4),time char(20));";
52       createTable(sql);                                  //执行创建表的 SQL 语句
53      }
54      public static void insertTime(int grade) {         //插入时间的方法
55       Date d=new Date();                                 //创建时间对象
56         String curr_time=(d.getYear()+1900)+"-"+(d.getMonth()+1<10?"0"+
                                                            //时间的年要加 1900
57                (d.getMonth()+1):(d.getMonth()+1))+"-"+d.getDate()+"-"+
58                d.getHours()+"-"+d.getMinutes()+"-"+d.getSeconds();
59       String sql_insert="insert into paihangbang values("+grade+","+"'"+curr_
         time+"');";
60       insert(sql_insert);                                //调用插入数据方法插入数据
61      }}
```

- 第5-13行为连接数据库方法，在该方法中，如果数据库已经创建，则只是打开连接；如果没有创建，则先创建然后打开连接。
- 第32-49行为根据SQL语句查询得到结果集，然后将结果集数据存入向量，每一行有两列，即时间列和成绩列，而每一行的数据需存一个向量中，然后将每一个行向量存入总结果向量中。最后将结果向量返回以供排行榜界面显示。
- 第54-61行为插入一行数据，首先得到当前时间，然后根据时间和分数构造SQL语句，最后调用插入数据方法插入数据。

21.6　游戏界面相关类

前面已经对各个辅助界面类进行了介绍，从本小节开始将进入游戏开发最主要的环节，即游戏界面开发环节。本小节先对整个游戏界面类进行介绍，然后再详细介绍各个与游戏界面相关类的开发，具体步骤如下。

21.6.1　游戏界面绘制类GLGameView

该类为整个游戏的绘制类，是整个游戏的呈现类，也是整个游戏逻辑计算类和物理世界模拟控制类。开发该类的详细步骤如下。

（1）首先开发的是GLGameView类的框架，先理解该类的框架有助于读者对本类的整体结构的理解，具体代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的GLGameView.java。

```
package com.bn.tl;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘的源代码
```

```java
public class GLGameView extends GLSurfaceView{
……//此处省略了相关成员变量的声明，读者可自行查阅随书光盘的源代码
    public GLGameView(Context context) {                          //构造器
    ……//此处省略了构造器中的部分代码，将在后面的步骤中给出
    }
    public void initWorld(){                                      //初始化物理世界的方法
    ……//此处省略了初始化世界的部分代码，将在后面的步骤中给出
        }
        @Override
        public boolean onTouchEvent(MotionEvent e)  {    //屏幕监听方法
        ……//此处省略了屏幕触摸事件处理方法中的部分代码，将在后面的步骤中给出
        }
        public class SceneRenderer implements GLSurfaceView.Renderer{//渲染类
            public void initShader(){                             //初始化绘制物体的 shader 程序
            ……//此处省略了初始化绘制物体 shader 的部分代码，将在后面的步骤中给出
            }
            public void initShaderWelcome(){                      //初始化 3D 欢迎界面的 shader
                wr.intShader(ShaderManager.getCommTextureShaderProgram());
                dot.intShader(ShaderManager.getCommTextureShaderProgram());
            }
            @Override
            public void onDrawFrame(GL10 gl){
            ……//此处省略了绘制场景的部分代码，将在后面的步骤中给出
            }
            public void onDrawShiping(){                          //绘制视频界面上的各个按钮
            ……//此处省略了绘制帮助界面的部分代码，将在后面的步骤中给出
            }
            public void drawProcessBar(){                         //绘制加载界面
            ……//此处省略了绘制加载界面的部分代码，将在后面的步骤中给出
            }
            public void drawBasketboard(){                        //绘制篮板
            ……//此处省略了绘制篮板  的部分代码，将在后面的步骤中给出
            }
            public void drawDeshBoard(){                          //绘制仪表板
            ……//此处省略了绘制仪表板的部分代码，将在后面的步骤中给出
            }
            public void drawHouse(){                              //绘制整个房间
            ……//此处省略了绘制整个房间的部分代码，将在后面的步骤中给出
            }
            @Override
            public void onSurfaceChanged(GL10 gl, int width, int height){
            ……//此处省略了 onSurfaceChanged 方法中的部分代码，将在后面的步骤中给出
            }
            @Override
            public void onSurfaceCreated(GL10 gl, EGLConfig config){
            ……//此处省略了 onSurfaceCreated 方法中的部分代码，将在后面的步骤中给出
            }
            public void initTaskReal(){                           //加载 3D 纹理
            ……//此处省略了加载 3D 纹理的部分代码，将在后面的步骤中给出
            }
```

```
53          }
54          public void initObjectWelcome(Resources r) {     //创建欢迎界面的物体对象
55              wr=new HuanYingJieMianJuXing(SCREEN_WIDHT/SCREEN_HEIGHT*2,2);//背景
56              dot=new HuanYingJieMianJuXing(0.4f, 0.4f);  //旋转图片
57          }
58          public  void initObject(Resources r) {            //初始化对象方法
59          ……//此处省略了初始化对象的部分代码，将在后面的步骤中给出
60          }
61          public  void ballControlUtil(){                   //这里对球的物理引擎部分进行操作
62          ……//此处省略了对篮球处理的部分代码，将在后面的步骤中给出
63          }
64          public void initBitmap(){                      //这里主要是用于将图片加载成 Bitmap
65          ……//此处省略了部分代码，将在后面的步骤中给出
66          }
67          public Bitmap loadTexture(int drawableId)   {    //通过 IO 加载图片
68          ……//此处省略部分代码，将在后面的步骤中给出
69          }
70          public int initTexture(Bitmap bitmapTmp,boolean needRrelease){//初始化纹理
71          ……//此处省略了部分代码，将在后面的步骤中给出
72          }}
```

- 第8-10行为初始化物理世界的方法，由于该游戏使用第三方开源的物理引擎JBullet，在对场景进行模拟时，首先需要初始化物理世界和各个部件。
- 第11-14行为屏幕触摸事件处理方法，在该方法中主要对球进行操控，通过触摸球，然后触摸移动给予球初速度，详细情况将在后面的步骤中给出。
- 第16-18行为对3D物理世界绘制物体需要的着色器程序进行初始化。
- 第19-22行为初始化加载界面下两个矩形的着色器程序，除了在加载界面下绘制的矩形外，其他物体均是在进入3D界面后对其加载的，所以在切换到3D界面时，只对这两个矩形进行加载。
- 第23-26行为主要的绘制方法，该方法负责绘制3D界面中的所有物体，包括加载界面，游戏界面和视频播放界面。
- 第27-29行为绘制视频播放界面，即帮助界面，当玩家在主菜单界面下单击帮助按钮时，进入3D场景后绘制的是视频播放界面。
- 第30-32行为绘制加载界面，在主菜单界面下单击开始游戏按钮，进入3D场景，进入后首先绘制加载界面，然后再绘制方法中对其他物体进行初始化。
- 第33-41行为绘制篮板、房间和仪表盘的方法。
- 第42-49行为重写的两个方法，这两个方法在切换到3D界面时首先被调用，而着色器程序的初始化等工作只能放在这两个方法和onDrawFrame方法中，且不能在其他线程中初始化。

- 第50-52行为加载相应的纹理，加载纹理也和初始化着色器程序一样，需放在此3个方法中。
- 第54-72行为分别完成各个小功能的方法，此处就不再做过多的介绍，详细情况请见注释。

（2）完成该类代码框架的开发后，接下来开发的是该类的构造器，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的GLGameView.java。

```
public GLGameView(Context context) {
        super(context);
        father=(BasketBall_Shot_Activity)context;              //主 Activity
        cx=(float)(tx+Math.cos(Math.toRadians(xAngle))*        //计算摄像机的位置
             Math.sin(Math.toRadians(yAngle))*DISTANCE);        //摄像机 x 坐标
        cz=(float)(tz+Math.cos(Math.toRadians(xAngle))*
             Math.cos(Math.toRadians(yAngle))*DISTANCE);        //摄像机 z 坐标
        cy=(float)(ty+Math.sin(Math.toRadians(xAngle))*DISTANCE);//摄像机 y 坐标
        deadtimesMS=0;                                          //初始化倒计时
             flag=true;                                         //物理模拟线程运行标志位
             Constant.defen=0;                                  //得分清零
             this.setEGLContextClientVersion(2);                //设置渲染模式为 2.0
             myRenderer=new SceneRenderer();                    //创建渲染器
             setRenderer(myRenderer);                           //设置渲染器
             setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
        }
```

说明

此为该类的构造器，在构造器中首先计算摄像机的位置，然后初始化一些相关变量，最后创建渲染器，设置渲染器，使其渲染模式为主动渲染。

（3）完成该类构造器的开发后，接下来开发的是该类物理事件的初始化方法initWorld，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的GLGameView.java。

```
1 public void initWorld()    {
2         CollisionConfiguration collisionConfiguration =
3                 new DefaultCollisionConfiguration();        //创建碰撞检测配置信息对象
5     dispatcher = new CollisionDispatcher(collisionConfiguration); //创建碰撞检测算法分配者对象
6         BroadphaseInterface overlappingPairCache = new DbvtBroadphase();
7         SequentialImpulseConstraintSolver solver =          //创建推动约束解决者对象
8                         new SequentialImpulseConstraintSolver();
9         shijie = new DiscreteDynamicsWorld(dispatcher,      //创建物理世界对象
10                            overlappingPairCache, solver, collisionConfiguration);
11            shijie.setGravity(new Vector3f(0, G, 0));       //设置重力加速度
12            lifangti=new BoxShape(new                       //创建共用的立方体
13            Vector3f(36*LANBAN_BILIXISHU/2,21*LANBAN_BILIXISHU/2,LANBAN_BILIXISHU/2));
15            pingmian=new CollisionShape[]{                      //创建共用的平面形状
16                new StaticPlaneShape(new Vector3f(0, 1f, 0f), 0.05f),//地面
17                new StaticPlaneShape(new Vector3f(0, 0, -1), 0),//前面
18                new StaticPlaneShape(new Vector3f(0, 0, 1), 0), //后面
19                new StaticPlaneShape(new Vector3f(1, 0, 0), 0), //左面
20                new StaticPlaneShape(new Vector3f(-1, 0, 0), 0),//右面
21            };
22            basketballShape=new SphereShape(QIU_R);             //创建篮球刚体
23            lanquanjiaonang=new CapsuleShapeZ(ZJ_R,     QIU_R*   //创建搭建篮圈的胶囊
24                    (float)(Math.cos(Math.toRadians((180-360/jiaolanggeshu)/2)))*
                      2-ZJ_R*2);
25        }
```

- 第2-11行为创建世界信息，物理世界对象，设置物理世界加速度等。
- 第12-24行为创建该游戏中需要使用的物体形状，该游戏中用到的物理形状有立方体、平面、球和胶囊，其中立方体用来模拟篮板，胶囊用来搭建篮圈。

（4）完成initWorld方法的开发后，接下来开发的是该类屏幕触摸事件的处理方法onTouchEvent，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的GLGameView.java。

```java
@Override
    public boolean onTouchEvent(MotionEvent e)  {
          float y = e.getY();                                    //得到触摸点的坐标
          float x = e.getX();
          boolean isDianjibb=false;                              //是否单击到篮球
          switch (e.getAction()){
          case MotionEvent.ACTION_DOWN:                          //触摸按下
            if(isnoHelpView){                                    //如果是视频播放界面
                 if(x>0&&x<96*ratio_width&&y>704*ratio_height){
                          for(int i=0;i<3;i++){                  //循环将球摆放在初始位置
                          Transform tt=new Transform();  //Transform 对象
                           tt.origin.set(new Vector3f(STARTBALL[(i)%3][0],
                          STARTBALL[(i)%3][1],STARTBALL[(i)%3][2]));//设置位置坐标
                           ballLst.get((i)%3).body.setCenterOfMassTransform(tt);
                          }
                          isnoHelpView=false;                    //界面设置为游戏界面
                          isnoPlay=true;
                          defen=1;                               //得分起始为 1 分
                     }
                     else if(x>384*ratio_width&&y>704*ratio_height){
                          isnoPlay=!isnoPlay;                    //单击处为播放或者暂停按钮处
                     }
                     return true;
                 }
           ArrayList<BasketBallForDraw> ballLstt=new ArrayList<BasketBallForDraw>();
                     for(BasketBallForDraw ball:ballLst){    //复制列表
                          ballLstt.add(ball);
                     }
                 touch_x=x;                                          //记录按下的位置
                 touch_y=y;
             float x3d=CHANGJING_WIDTH*touch_x/SCREEN_WIDHT-0.5f*CHANGJING_WIDTH;
             float y3d=CHANGJING_HEIGHT*(SCREEN_HEIGHT-touch_y)/SCREEN_HEIGHT;
                  for(BasketBallForDraw ball:ballLstt){       //循环篮球列表
                     float ball_x=ball.body.getWorldTransform(new Transform()).origin.x;
                     float ball_y=ball.body.getWorldTransform(new Transform()).origin.y;
                     float ball_z=ball.body.getWorldTransform(new Transform()).origin.z;
                       float ball_scale=1.5f*QIU_R;                  //篮球的半径
                     if(x3d<ball_x+ball_scale&&x3d>ball_x-ball_scale&&
                          y3d<ball_y+ball_scale&&y3d>ball_y-ball_scale&&
                          ball_z>1.55f){                             //如果单击处是篮球的位置
                          curr_ball=ball;                            //记录触摸到的篮圈
                          break;                                     //跳出此循环
```

```
            }}
          if(curr_ball==null){                            //如果单击处都不是球的位置
              for(BasketBallForDraw ball:ballLst){  //循环察看所有球是否为静止
                  if(!ball.body.wantsSleeping()){  //只要有一个还不是静止的
                      isDianjibb=true;     //说明玩家就不能变动摄像机位置
          }}}
          if(!isDianjibb&&curr_ball==null){ //如果所有球都是静止的，并且为触摸球
             isnoCamear=true;            //可通过触摸移动变动摄像机位置
          }
         break;
       case MotionEvent.ACTION_UP:                  //触摸抬起事件处理
         float dx=x-touch_x;                        //x 方向上的移动距离
         float max_fingerTouch = 110*ratio_height;//这里设置手指最大的触摸距离
float dy=(y-touch_y)>0?0:((y-touch_y)<-max_fingerTouch?-max_fingerTouch:
(y-touch_y));
          isnoCamear=false;                         //视角开始向回滚
          isDianjibb=false;
          if(curr_ball!=null)                       //如果触摸到球了
             float vTZ=1f;                          //速度调整值
             Vector3f linearVelocity=curr_ball.body.getLinearVelocity(new
             Vector3f());
          if(linearVelocity.x>-vTZ&&linearVelocity.x<vTZ&&
                                                    //看其当前速度是否在调整值内
             linearVelocity.y<vTZ&&linearVelocity.y>-vTZ&&
                                                    //是则可对其进行赋予初速度
                linearVelocity.z>-vTZ&&linearVelocity.z<vTZ) {
              float vx=dx*10/SCREEN_WIDHT;//根据滑动的距离计算对应方向的速度
                    float vy=-dy*76*vFactor/SCREEN_HEIGHT;
                    float vz=dy*28/SCREEN_HEIGHT;
                    curr_ball.body.activate();//唤醒该球
                curr_ball.body.setLinearVelocity(new Vector3f(vx,vy,vz));
                                                    //设置线速度
                   curr_ball.body.setAngularVelocity(new Vector3f(5,0,0));
                                                    //设置角速度
                    curr_ball=null;                 //将该应用设为空
             }}
          curr_ball=null;                           //将该应用设为空
         break;
       case MotionEvent.ACTION_MOVE:                //如果是移动事件
         if(isnoCamear) { //如果其为单击时所有球都停止，并且为触摸移动屏幕改变视角
             float ddy = y - mPreviousY;           //计算触控笔 Y 位移
              xAngle += ddy * TOUCH_SCALE_FACTOR;//方位角改变
              if(xAngle<0) {                        //仰角小于 0 则仰角强制为 0
                 xAngle=0;
              }
              if(xAngle>35) {                //当摄像机仰角大于 35,将其强制为 35
                 xAngle=35;
              }
         cx=(float)(tx+Math.cos(Math.toRadians(xAngle))*Math.sin(
```

```
86                  Math.toRadians(yAngle))*DISTANCE); //摄像机x坐标
87          cz=(float)(tz+Math.cos(Math.toRadians(xAngle))*Math.cos(
88                  Math.toRadians(yAngle))*DISTANCE); //摄像机z坐标
89          cy=(float)(ty+Math.sin(Math.toRadians(xAngle))*DISTANCE);//摄像机y坐标
90          }
91          break;
92        }
93        mPreviousY = y;                          //记录触控笔位置
94        mPreviousX = x;                          //记录触控笔位置
95        return true;
96    }
```

- 第7-52行为屏幕触摸按下事件的处理，如果当前界面为视频播放界面，则查看是否单击的是播放/暂停按钮或停止按钮。如果不是视频播放界面，则查看其是否是触摸到一个静止的球，如果是则记录该球，如果没有触摸球，并且所有球都是静止的，则说明玩家要触摸改变摄像机位置。
- 第53-74行为屏幕触摸抬起事件的处理，如果记录触摸到的球的引用不为空，则说明玩家在触摸按下时，触摸到了一个静止的球，则根据玩家触摸移动的距离计算对应方向的速度，然后将此速度赋予该球即可。
- 第75-91行为屏幕触摸移动事件的处理，如果改变摄像机位置的标志位为true，说明玩家在按下屏幕时，所有球都静止并且没有触摸到球。此时则根据其移动的距离来计算摄像机位置的改变，以此来改变视角观看游戏场景。

（5）完成onTouchEvent方法的开发后，接下来开发的是渲染器类中的初始化着色器程序方法initShader，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/t1目录下的GLGameView.java。

```
public void initShader(){
        ball.initShader(ShaderManager.getShadowshaderProgram());               //球
        lanWuang.initShader(ShaderManager.getBasketNetShaderProgram());        //篮网
        backboard.initShader(ShaderManager.getCommTextureShaderProgram());     //篮板
        zj.initShader(ShaderManager.getLigntAndTexturehaderProgram());         //支架
        lankuang. initShader (ShaderManager.getLigntAndTexturehaderProgram()); //篮筐
        downPanel. initShader (ShaderManager.getCommTextureShaderProgram()); //地面
        frontPanel. initShader (ShaderManager.getCommTextureShaderProgram());//后面
        leftPanel. initShader (ShaderManager.getCommTextureShaderProgram()); //左面
            ybb.initShader(ShaderManager.getCommTextureShaderProgram());     //仪表板
            shuzi.initShader(ShaderManager.getBlackgroundShaderProgram());   //数字
            if(isnoHelpView){                                                //如果是帮助界面
            shipingjimian.initShader(ShaderManager.getCommTextureShaderProgram());
            wenziJuxing. initShader (ShaderManager.getCommTextureShaderProgram());
                                                                             //文字矩形
            }}
```

> **说明**
>
> 该方法中分别调用了篮球、篮板、篮网等游戏实体类对象的initShader方法，初始化了其相应的着色器代码脚本。

（6）完成initShader方法的开发后，接下来开发的是渲染器类中的主绘制方法onDrawFrame，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的GLGameView.java。

```
@Override
    public void onDrawFrame(GL10 gl)   {
     GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT |          //清除深度缓冲与颜色缓冲
                    GLES20.GL_COLOR_BUFFER_BIT);
     GLES20.glEnable(GLES20.GL_DEPTH_TEST);                //打开深度检测
     if(!hasLoadOk) {                                      //如果是加载界面
         MatrixState.pushMatrix();                         //保护现场
       MatrixState.setProjectOrtho(-ratio, ratio, -1, 1, 1, 10);//设置正交投影矩阵
       MatrixState.setCamera(0, 0, 0, 0, 0,-1, 0, 1, 0);         //设置摄像机位置
           MatrixState.pushMatrix();                       //保护现场
           MatrixState.translate(0, 0, -2);                //沿着 z 轴负方向推动 2 个单位
           MatrixState.rotate(90, 1, 0, 0);                //绕 X 轴旋转 90°
           wr.drawSelf(welcomeid);                         //绘制加载界面背景
           MatrixState.popMatrix();                        //恢复现场
           drawProcessBar();                               //绘制旋转矩形
           MatrixState.popMatrix();                        //恢复现场
           if(isFirst) {                                   //如果是第一帧
           isFirst=false;                                  //标记已经显示完第一帧了
           } else{                                    //如果不是第一帧则开始加载其他物体
           initTaskReal();                                 //加载其他物体
           curr_process++;                                 //加载一次此标记自加一
           if(curr_process>7) {                            //当所有物体加载完毕
               hasLoadOk=true;                             //加载完毕标志位置为 true
               start=System.currentTimeMillis();           //记录加载完毕时间
           }}}
      else if(System.currentTimeMillis()-start>7000) {//当前时间到加载完毕的时间差有 7s
        MatrixState.setProjectFrustum(-ratio, ratio, -1, 1, 4f, 100);
       MatrixState.setCamera(cx, cy, cz, tx, ty,tz, 0, 1, 0); //调用此方法设置摄像机位置
        MatrixState.pushMatrix();                          //保护现场
        drawHouse();                                       //绘制整个场景
        MatrixState.setLightLocation(3, CHANGJING_HEIGHT*1.7f, 5); //初始化光源位置
        drawBasketboard();                                 //绘制篮架
        GLES20.glEnable(GLES20.GL_BLEND);                  //开启混合，设置混合因子
      GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
        ArrayList<BasketBallForDraw> ballLstt=new ArrayList<BasketBallForDraw>();
            for(BasketBallForDraw ball:ballLst){    //复制列表
                ballLstt.add(ball);
            }
         for(BasketBallForDraw bf:ballLstt) {    //绘制所有球在地板和后面墙上的影子
             bf.drawSelf(basketbalolid,1,0,0);     //地板影子
             bf.drawSelf(basketbalolid,1,3,0);     //后面影子
         }
         GLES20.glDisable(GLES20.GL_BLEND);                //关闭混合
         for(BasketBallForDraw bf:ballLstt) {
```

```
            bf.drawSelf(basketbalolid,0,0,0);         //绘制所有球
        }
        GLES20.glEnable(GLES20.GL_DEPTH_TEST);        //开启深度检测
        GLES20.glEnable(GLES20.GL_BLEND);             //开启混合，设置混合因子
    GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
        for(BasketBallForDraw bf:ballLst) {           //循环绘制所有球在篮板上的影子
            bf.drawSelf(basketbalolid,1,4,1);         //篮板上的影子
        }
            MatrixState.pushMatrix();                 //绘制篮网
            MatrixState.translate(LANBAN_X, LANBAN_Y-LANQIU_HEIGHT/6,
            LANBAN_Z+ZJ_LENGTH+LANKUANG_R);           //推动坐标系到篮网位置
          MatrixState.translate(0, -LANWANG_H-LANKUANG_JM_R, 0);
          lanWuang.drawSelf(lanwangId,                //绘制篮网
                lanWangRaodon);
          MatrixState.popMatrix();                    //恢复现场
          GLES20.glDisable(GLES20.GL_BLEND);          //关闭混合
             drawDeshBoard();                         //绘制仪表盘
             MatrixState.popMatrix();                 //恢复现场
             onDrawShiping();                         //绘制播放界面
         }else {                                      //如果等待的时间没有 7s
          curr_process++;                             //自加
          MatrixState.pushMatrix();                   //保护现场
         MatrixState.setProjectOrtho(-ratio, ratio, -1, 1, 1, 10);//设置产生正交投影
          MatrixState.setCamera(0, 0, 0, 0, 0,-1, 0, 1, 0); //设置摄像机的位置
          MatrixState.pushMatrix();                   //保护现场
          MatrixState.translate(0, 0, -2);            //推动物体坐标到对应位置
          MatrixState.rotate(90, 1, 0, 0);            //设置绕 x 轴旋转 90°
          wr.drawSelf(welcomeid);                     //绘制背景
          MatrixState.popMatrix();                    //恢复现场
          drawProcessBar();                           //绘制旋转矩形
          MatrixState.popMatrix();                    //恢复现场
          if(!isStart&&System.currentTimeMillis()-start>6890) {
                                                      //如果时间到超过并是第一次
          isStart=true;                               //游戏开始标志设置为 true
          father.curr=WhichView.GAME_VIEW;            //当前界面为游戏界面
           ArrayList<BasketBallForDraw> ballLstt=new ArrayList<BasketBallFor
           Draw>();
           for(BasketBallForDraw ball:ballLst){       //复制列表
                ballLstt.add(ball); }
          if(isnoHelpView){                           //如果是帮助界面
          float vx=STARTBALL_V[0][0];                 //得到对应球的速度保证其能进球
           float vy=STARTBALL_V[0][1];
           float vz=STARTBALL_V[0][2];
           ballLstt.get(0).body.activate();           //唤醒刚体
           ballLstt.get(0).body.setLinearVelocity(new Vector3f(vx,vy,vz));
                                                      //设置线速度
           ballLstt.get(0).body.setAngularVelocity(new Vector3f(5,0,0));
                                                      //设置角速度
           startY=0;                                  //起始坐标归 0
```

```
90         new Thread(){                                  //新建一个线程来模拟投球
91        int array_id=1;                                 //发射第几个球
92        boolean isnoFashe=false;                        //是否开始进行手的动画模拟
93        @Override
94        public void run(){                              //重新 run 方法
95          ……//此处省略了该方法中的部分代码将在后面的步骤中给出
96          }}.start();                                   //启动线程
97         }else{                                         //如果不是视频播放界面
98                 float vx=0;                            //一开始时，将中间球给予一个初速度
99                 float vy=10.1f*vFactor;
100                float vz=-3.0f;
101                ballLstt.get(1).body.activate();
102            ballLstt.get(1).body.setLinearVelocity(new Vector3f(vx,vy,vz));
103            ballLstt.get(1).body.setAngularVelocity(new Vector3f(5,0,0));
104        }}}}
```

- 第6-16行为在当系统资源加载完毕时，只绘制加载界面上的背景矩形和不断旋转的矩形，第17-25行为在第一帧时不对物体进行加载，从而使得屏幕不出现黑屏效果。
- 第26-64行为当加载完毕7s后，才开始绘制整个游戏界面，此场景不管是在游戏界面还是帮助界面都得绘制。之所以要这么多时间，是因为初始化物理世界需要一定的时间。
- 第64-104行为当此时与加载完毕的时间相差未达到7s时，只绘制加载界面。如果当前是视频播放界面时，在即将达到7s时，给第一个球一个初速度，使其在切换到游戏界面时刚好有一个动作，然后新建一个线程来模拟视频播放，即循环地给予每一个球初速度。而在视频播放界面时，也在时间快达到7s时，给中间球一个初速度。

（7）完成onDrawFrame方法的开发后，接下来开发的是在onDrawFrame方法下新建的线程中的run方法，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的GLGameView.java。

```
@Override
public void run(){
    while(isnoHelpView){                                          //如果是视频播放界面
        if(!isnoPlay){                                            //如果是暂停界面
            continue;                                             //结束此次循环
        }
        ArrayList<BasketBallForDraw> ballLstt=new ArrayList<BasketBallForDraw>();
        for(BasketBallForDraw ball:ballLst){                      //复制列表
        ballLstt.add(ball);
            }
            startY-=2;                                            //文字的位置
            startY%=1100;                                         //文字 y 坐标
            array_id%=3;                                          //列表下标
            shipingJs+=100;
            if(!isnoFashe&&(shipingJs)%5000==4000){               //每 5000 ms 一次
            Transform tt=new Transform();
            tt.origin.set(new Vector3f(STARTBALL[(array_id+2)%3][0],//设置位置
                    STARTBALL[(array_id+2)%3][1],STARTBALL[(array_id+2)%3][2]));
            ballLstt.get((array_id+2)%3).body.setCenterOfMassTransform(tt);

            shouX=STARTBALL[array_id][0]/2;                       //手抓的 x 坐标
            shouY=-0.9f;                                          //手抓 y 坐标
            isnoFashe=true;                                       //是否让手抓移动标志
            }
            if(isnoFashe){                                        //如果已经给予初速度
            if(array_id%3==0){                                    //如果是 0 号球
                shouX+=0.03f;                           //其位置每次自加一个值使其向右移动
            }else if(array_id%3==2){                              //若 2 号球则移动 x 坐标
                shouX-=0.03f;
            }
            shouY+=0.1f;                                          //y 方向每次自加一个值
            if(shouY>1.2f){
                shouY=5f;
            }
            }else{                                                //否则手抓不在屏幕上
            shouY=5f;
            }
            if((shipingJs)%5000==0){                              //如果每次间隔 5s
            isnoFashe=false;                                      //发射完毕后
            shouY=4;                                              //手抓不在屏幕上
            float vx=STARTBALL_V[array_id][0];//*10/SCREEN_WIDHT;//设置速度
                float vy=STARTBALL_V[array_id][1];
                float vz=STARTBALL_V[array_id][2];
                ballLstt.get(array_id).body.activate();           //唤醒对应球的刚体
               ballLstt.get(array_id).body.setLinearVelocity(new Vector3f(vx,vy,vz));
                                                                  //设置线速度
                ballLstt.get(array_id).body.setAngularVelocity(new Vector3f(5,0,0));
                                                                  //设置角速度
                array_id++;                                       //下一次给下一个球赋予速度
                }
                try {
                    Thread.sleep(100);                            //线程睡眠 100ms
                 } catch (InterruptedException e) {               //捕获异常
                  e.printStackTrace();                            //打印异常
                 }}}
```

- 第15-23行为每次在第4s时，开始播放手抓的动画，将上一次运动球的位置放到初始位置。
- 第24-36行为对手抓坐标的变化，来模拟用手指触摸球并移动来投球。
- 第37-47行为在间隔时间为5s时，给球一个初速度。

（8）完成线程中run方法的开发后，接下来开发的是渲染器类中绘制视频界面的方法onDrawShiping，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的GLGameView.java。

```
public void onDrawShiping(){                              //绘制视频界面上的各个按钮
        if(!isnoHelpView){                                //如果不是视频播放界面就不绘制
            return;
        }
    if(wenziId!=-1) {                                     //如果文字纹理还没加载则不绘制
        GLES20.glDeleteTextures(1, new int[]{wenziId}, 0);
```

```
        }
        Bitmap bm=Constant.generateWLT(Constant.content, wenziWidth, wenziHeight);
        wenziId=initTexture(bm,true);                           //根据图片生成纹理
            MatrixState.pushMatrix();                           //保护现场
            MatrixState.setProjectOrtho(-1, 1, -1, 1, 1, 10);   //设置成正交投影
            MatrixState.setCamera(0, 0, 0, 0, 0, -1, 0, 1, 0);  //设置摄像机
            GLES20.glEnable(GLES20.GL_BLEND);                   //开启混合,
            GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA,             //设置混合因子
                          GLES20.GL_ONE_MINUS_SRC_ALPHA);
            MatrixState.pushMatrix();                       //保护现场
            MatrixState.translate(0,-0.05f, -2f);           //推动到一定位置
            MatrixState.rotate(90, 1, 0, 0);                //绕 x 轴旋转 90°
            wenziJuxing.drawSelf(wenziId);                  //绘制文字矩形
            MatrixState.popMatrix();                        //恢复现场
             MatrixState.pushMatrix();                      //保护现场
            MatrixState.translate(-0.8f, -0.88f, -2f);      //推动到一定位置
            MatrixState.rotate(90, 1, 0, 0);                //绕 X 轴旋转 90°
            shipingjimian.drawSelf(stopId);                 //绘制停止按钮
            MatrixState.popMatrix();                        //恢复现场
            MatrixState.pushMatrix();                       //保护现场
            MatrixState.translate(0.8f, -0.88f, -2f);       //推动到一定位置
            MatrixState.rotate(90, 1, 0, 0);                //绕 X 轴旋转 90°
            if(isnoPlay){                                   //如果在播放中绘制暂停按钮
            shipingjimian.drawSelf(pauseId);                //绘制暂停按钮
            }else{                                          //否则绘制播放按钮
            shipingjimian.drawSelf(playId);                 //绘制播放按钮
            }
            MatrixState.popMatrix();                        //恢复现场
            MatrixState.pushMatrix();                       //保护现场
            MatrixState.translate(shouX,shouY, -1.8f);      //设置手抓的位置
            MatrixState.rotate(90, 1, 0, 0);                //绕 x 旋转 90°
            shipingjimian.drawSelf(shouId);                 //绘制手按钮
            MatrixState.popMatrix();                        //恢复现场
            GLES20.glDisable(GLES20.GL_BLEND);              //关闭混合
            MatrixState.popMatrix();                        //恢复现场
             }
```

说明

该方法为在视频播放界面时，实时生成一幅绘制了文字的图片，然后将该图片初始化成纹理，最后将该纹理图片绘制在界面上，以达到文字向上滚动的效果。对于渲染器类后面的几个具体的绘制方法，其本质没有多大区别，此处不再对其详细介绍，请读者自行查阅随书光盘中的源代码。

（9）完成onDrawShiping方法的开发后，接下来开发的是各个物体加载方法initTaskReal，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的GLGameView.java。

```
public void initTaskReal(){
        initBitmap();                                          //加载图片
        if(curr_process==2){
            defen=0;                                           //还原得分
            daojishi=60;                                       //还原倒计时
            SHENGYING_FLAG=SOUND_MEMORY;                       //还原声音选择
            DEADTIME_FLAG=true;                                //开启倒计时
            initObject(GLGameView.this.getResources());        //创建各个物体
            ShaderManager.compileShaderReal();                 //加载 shader 程序
            }
            if(curr_process==3){
                initShader();                                  //初始化 shader 程序
            }
            if(curr_process==4){
                dibanTexId=initTexture(bm_floor,true);         //地板纹理
                zuobianQiangID=initTexture(bm_swall1,true); //左墙纹理
                youbianQiangID=initTexture(bm_swall3,true); //右墙纹理
                houmianQiangID=initTexture(bm_swall2,true); //后墙纹理
                basketbalolid=initTexture(bm_basketball,true);  //篮球纹理
                lanbanId=initTexture(bm_lanban2,true);         //篮板纹理
                shijianxiansBeijingId=initTexture(bm_yibiaoban,true);
                                                               //顶上时间等显示纹理
                shuziId=initTexture(bm_number,true);           //数字 bm_basketnet
                lanwangId=initTexture(bm_basketnet,true);      //篮网纹理
                if(isnoHelpView){
                    shouId=initTexture(bm_shou,true);;         //抓
                    stopId=initTexture(bm_stop,true);;         //停止按钮
                    pauseId=initTexture(bm_pause,true);        //暂停按钮
                    playId=initTexture(bm_play,true);          //播放按钮
                }}}
```

```
        initBitmap();                                           //加载图片
        if(curr_process==2){
            defen=0;                                            //还原得分
            daojishi=60;                                        //还原倒计时
            SHENGYING_FLAG=SOUND_MEMORY;                        //还原声音选择
            DEADTIME_FLAG=true;                                 //开启倒计时
            initObject(GLGameView.this.getResources());         //创建各个物体
            ShaderManager.compileShaderReal();                  //加载 shader 程序
            }
            if(curr_process==3){
                initShader();                                   //初始化 shader 程序
            }
            if(curr_process==4){
                dibanTexId=initTexture(bm_floor,true);          //地板纹理
                zuobianQiangID=initTexture(bm_swall1,true); //左墙纹理
                youbianQiangID=initTexture(bm_swall3,true); //右墙纹理
                houmianQiangID=initTexture(bm_swall2,true); //后墙纹理
                basketbalolid=initTexture(bm_basketball,true);  //篮球纹理
                lanbanId=initTexture(bm_lanban2,true);          //篮板纹理
                shijianxiansBeijingId=initTexture(bm_yibiaoban,true);
                                                                //顶上时间等显示纹理
                shuziId=initTexture(bm_number,true);            //数字 bm_basketnet
                lanwangId=initTexture(bm_basketnet,true);       //篮网纹理
                if(isnoHelpView){
                    shouId=initTexture(bm_shou,true);;          //抓
                    stopId=initTexture(bm_stop,true);;          //停止按钮
                    pauseId=initTexture(bm_pause,true);         //暂停按钮
                    playId=initTexture(bm_play,true);           //播放按钮
                }}}
```

说明

该方法为加载各个3D界面需要的物体，有对图片的加载，各个物体对象的创建，着色器程序的加载和初始化，还有各个纹理id的初始化等，详细情况请见注释。

（10）完成initTaskReal方法的开发后，接下来开发的是初始化对象方法initObject，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的GLGameView.java。

```
public  void initObject(Resources r)    {                       //初始化对象方法
        initWorld();                                            //初始化物理世界
        lankuang=MoXingJiaZai.loadFromFileVertexOnly("lankuang.obj", r);//篮筐
        ball=new BasketBallTextureByVertex(QIU_R);              //创建球
        lanWuang=new LanWang(LANKUANG_R,0.55f*LANKUANG_R,LANWANG_H,8);
        downPanel=new WenLiJuXing(CHANGJING_WIDTH, CHANGJING_LENGTH+1); //下
        frontPanel=new WenLiJuXing(CHANGJING_WIDTH, CHANGJING_HEIGHT);   //前
        leftPanel=new WenLiJuXing(CHANGJING_LENGTH, CHANGJING_HEIGHT);   //左
        backboard=new LanBan(                                   //创建篮板
                   LANBAN_BILIXISHU/2,
```

```
            36*LANBAN_BILIXISHU,21*LANBAN_BILIXISHU, r);
    zj=new Yuanzhu(ZJ_LENGTH, ZJ_R, 30, 1, r);          //篮球支架
    shuzi=new HuiZhiShuZi(r);                           //创建数字
    ybb=new WenLiJuXing(YBB_WIDTH,YBB_HEIGHT);          //创建仪表板
    ballLst.clear();                                    //清空列表
    ballLst.add (new BasketBallForDraw(ball, basketballShape, shijie, 1,
      STARTBALL_1[0],STARTBALL_1[1],STARTBALL_1[2],GROUP_BALL1,MASK_BALL1));
    ballLst.add (new BasketBallForDraw(ball, basketballShape, shijie, 1,
      STARTBALL_2[0],STARTBALL_2[1],STARTBALL_2[2],GROUP_BALL2,MASK_BALL2));
    ballLst.add (new BasketBallForDraw(ball, basketballShape, shijie, 1,
      STARTBALL_3[0],STARTBALL_3[1],STARTBALL_3[2],GROUP_BALL3,MASK_BALL3));
    planeS=new TianjiaBody[]{                          //将各个物体添加进物理世界
        new TianjiaBody(pingmian[0], shijie, 1, 0, 0, 0,1,1), //地面
        new TianjiaBody(pingmian[1], shijie, 1, 0,           //前面
              CHANGJING_HEIGHT/2, CHANGJING_LENGTH/2+QIU_R*2,0,0),
        new TianjiaBody(pingmian[2], shijie, 1, 0,           //后面
              CHANGJING_HEIGHT/2, -CHANGJING_LENGTH/2,1,1),
        new TianjiaBody(pingmian[3], shijie, 1, -CHANGJING_WIDTH/2,
              CHANGJING_HEIGHT/2, 0,1,1),        //左面
        new TianjiaBody(pingmian[4], shijie, 1, CHANGJING_WIDTH/2,
              CHANGJING_HEIGHT/2, 0,1,1),        //右面
        new TianjiaBody(lifangti, shijie, 1, LANBAN_X, LANBAN_Y, LANBAN_Z,1,1),
                                                 //篮板
    };
    for(int i=0;i<jiaolanggeshu;i++){                  //创建篮筐胶囊，使其围成一个圆环
      langquanJiaonang[i]=new JiaoNangTianjiaBody(lanquanjiaonang, shijie, 0,
      LANBAN_X+LANKUANG_R*(float)(Math.cos(Math.toRadians(i*360/jiaolanggeshu))),
              LANBAN_Y-LANQIU_HEIGHT/6,
              LANBAN_Z+ZJ_LENGTH+LANKUANG_R+LANKUANG_R*(float)(
              Math.sin(Math.toRadians(i*360/jiaolanggeshu)))+ZJ_R,
              1,1,0,-(360/jiaolanggeshu/2)-360/jiaolanggeshu*i,0);
    }
    if(isnoHelpView){                                  //如果是视频界面
      shipingjimian=new WenLiJuXing(0.3f,0.18f); //创建视频播放界面按钮矩形
      wenziJuxing=new WenLiJuXing(1.8f,1.45f);   //创建文字矩形
    }
    new Thread(){                                      //创建物理模拟线程
      public void run(){
          while(flag) {
              try {
              if(isnoHelpView&&!isnoPlay){  //如果是播放视频界面，则不进行物理模拟
                  continue;
              }
               shijie.stepSimulation(1f/60.f,5);
               ballControlUtil();              //判断篮球是否进球和碰撞
               if(!isnoHelpView){              //如果不是视频播放界面，时间才改变
                   deadtimesMS+=10;
               }
               if(deadtimesMS>=daojishi*1000) {         //如果游戏时间到了
                   SQLiteUtil.insertTime(Constant.defen); //将分数记录进数据库
                   flag=false;                          //现场停止
```

```
61                      father.shengyinBoFang(2, 0);  //播放游戏结束声音
62                      father.xiaoxichuli.sendEmptyMessage(JIESHU_JIEMIAN);
63                  }
64                  if(xAngle>0&&!isnoCamear) {         //如果摄像机位置被触摸移动
65                      xAngle-=CAMERA_Y_SK_FH;         //恢复摄像机位置
66                      cx=(float)(tx+Math.cos(Math.toRadians(xAngle))*Math.sin(
67                              Math.toRadians(yAngle))*DISTANCE);//摄像机x坐标
68                      cz=(float)(tz+Math.cos(Math.toRadians(xAngle))*Math.cos(
69                              Math.toRadians(yAngle))*DISTANCE);//摄像机z坐标
70                      cy=(float)(ty+Math.sin(
71                              Math.toRadians(xAngle))*DISTANCE);//摄像机y坐标
72                      }
73                  if(xAngle<0&&!isnoCamear){          //如果xAngle<0则恢复摄像机位置
74                      xAngle+=CAMERA_Y_SK_FH;         //每次加一个角度
75                      cx=(float)(tx+Math.cos(Math.toRadians(xAngle))*Math.sin(
76                          Math.toRadians(yAngle))*DISTANCE);//摄像机x坐标
77                     cz=(float)(tz+Math.cos(Math.toRadians(xAngle))*Math.cos(
78                          Math.toRadians(yAngle))*DISTANCE);//摄像机z坐标
79                     cy=(float)(ty+Math.sin(Math.toRadians(xAngle))*DISTANCE);
80                  }
81                     Thread.sleep(10);                //线程休息10 ms
82                     }catch (Exception e) {           //捕获异常
83                         e.printStackTrace();         //打印异常
84                     }}}}.start();                    //启动线程
85          }
```

- 第3-21行为创建各个用于绘制3D世界的物体对象，并将代替篮球刚体的球添加进物理世界。
- 第22-41行为将各个对应物体的刚体添加进物理世界，其中第34～41行为用几个胶囊围成一个圆环来作为篮筐，因为JBullet中没有圆环。
- 第46-84行为新建一个线程来进行物理世界的模拟，物理世界会根据当前世界中的各个刚体，给出每个刚体的位置和旋转角度等，在每次绘制时，直接从对应刚体获取其位置和旋转角度绘制即可，同时该线程还进行计时操作，判断篮球是否进入篮筐，碰撞时播放对应的声音等。

（11）完成initObject方法的开发后，接下来开发的是判断篮球和各个物体的碰撞情况及处理方法ballControlUtil，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/t1目录下的GLGameView.java。

```
public void ballControlUtil(){
 for(BasketBallForDraw bf:ballLst) {                          //循环查看其碰撞情况,
        if(bf.body.isActive()&&SYSUtil.isCollided(shijie, bf.body,planeS[0].gangti)) {
            bf.body.applyForce(new Vector3f(0,0,60), new Vector3f(0,0,0));
        }
 Transform transform = bf.body.getMotionState().getWorldTransform(new Transform());
            float position_Y=transform.origin.y;              //获取篮球在世界坐标中的位置
            float position_X=transform.origin.x;
            float position_Z=transform.origin.z;
                Vector3f linearVelocity=bf.body.getLinearVelocity(new Vector3f());
               Vector3f angularVelocity=bf.body.getAngularVelocity(new Vector3f());
                                                              //旋转速度
```

```
            float linearVelocityDomain=0.08f;             //线性速度阈值
            float angularVelocityDomain=1.45f;            //角速度阈值
    float allLinearV=Math.abs(linearVelocity.x)+
        Math.abs(linearVelocity.y)+Math.abs(linearVelocity.z);
    float allAngularV=Math.abs(angularVelocity.x)+
        Math.abs(angularVelocity.y)+Math.abs(angularVelocity.z);
    if(bf.body.isActive()&&position_Y<STARTBALL_3[1]+0.05){//给予该球一个z方向的速度
    bf.body.setLinearVelocity(new Vector3f(0,linearVelocity.y,linearVelocity.z));
}
    if(bf.body.isActive()&&allLinearV<linearVelocityDomain&&allAngularV<
        angularVelocityDomain
        &&position_Y<STARTBALL_3[1]+0.05){            //当篮球位于最前方时使其静止
            bf.body.setActivationState(CollisionObject.WANTS_DEACTIVATION);
        }
            float lankuang_X=LANBAN_X;                //这里获取篮筐的中心位置坐标值
            float lankuang_Y=LANBAN_Y-LANQIU_HEIGHT/4;
            float lankuang_Z=LANBAN_Z+ZJ_LENGTH+LANKUANG_R;
            float lankuang_Radius=LANKUANG_R;         //篮筐的半径
            float ball_Radius=QIU_R;                  //篮球的半径
        float temp_distance=(float) Math.sqrt((position_X-lankuang_X)*(position
        _X-lankuang_X)+
            (position_Z-lankuang_Z)*(position_Z-lankuang_Z));
                                                      //计算篮球与篮筐直线的距离
            if(linearVelocity.y<0&&temp_distance<(lankuang_Radius-ball_Radius)
                &&position_Y>lankuang_Y) {            //如果篮球刚好位于篮筐中
                bf.ball_State=1;                      //标记一下
            }
            if(bf.ball_State==1&&position_Y<lankuang_Y&&
        position_X>lankuang_X-LANKUANG_R&&position_X<lankuang_X+LANKUANG_R&&
        position_Z>lankuang_Z-LANKUANG_R&&position_Z<lankuang_Z+LANKUANG_R) {
                Constant.defen++;                         //说明进入篮筐，得分数加一
                new Thread(){                             //新建一个线程使篮网扰动
                    @Override
                    public void run(){
                        int lanWang=0;                    //篮网扰动ID
                        int zTime=0;                      //时间
                        while(zTime<360){
                            lanWang++;                    //扰动ID自加
                            lanWangRaodon=lanWang%4;      //扰动ID在0~3内
                            zTime+=60;                    //时间自加
                            try {
                                Thread.sleep(60);         //线程休息
                            } catch (InterruptedException e) {//异常处理
                                e.printStackTrace();
                            }}
                        lanWangRaodon=0;
                    }}.start();                           //启动线程
                bf.ball_State=0;
                father.shengyinBoFang(3, 1);              //播放进球声音
            }
            Vector3f vt;
```

```
61          vt=bf.body.getLinearVelocity(new Vector3f());    //得到篮球的线速度
62          if((vt.y>1f&&vt.y<6.5f)                          //判断是否和地面撞上
63                  &&SYSUtil.isCollided(shijie,bf.body,planeS[0].gangti)) {
64              father.shengyinBoFang(1, 0);                 //播放碰撞声音
65          }else if (vt.x>1f&&SYSUtil.isCollided(shijie,bf.body,planeS[3].gangti)) {
66                  father.shengyinBoFang(1, 0);             //播放碰撞声音
67          }else if (vt.x<-1&&SYSUtil.isCollided(shijie,bf.body,planeS[4].gangti)) {
68                  father.shengyinBoFang(1, 0);             //播放碰撞声音
69          }else if (vt.z>1&&SYSUtil.isCollided(shijie,bf.body,planeS[2].gangti)) {
70                  father.shengyinBoFang(1, 0);             //播放碰撞声音
71          }
72          if(SYSUtil.isCollided(shijie,bf.body,planeS[5].gangti)) {
                                                             //判断是否和篮板撞上
73              if((Math.abs(vt.x)+Math.abs(vt.y)+Math.abs(vt.z))>2) {
74                      father.shengyinBoFang(1, 0);         //播放碰撞声音
75              }
76              bf.isnoLanBan=1;                             //已经碰撞过了标记
77          }else{
78              bf.isnoLanBan=0;                             //恢复标记
79          } }}
```

- 第3-25行为先判断其是否和地面相碰，如果相碰就给予该球z方向的一个冲量，使其滚到边沿，然后判断其是否在离边沿位置的距离小于一个阀值，如果在一定范围内则给其一个沿z方向的速度，如果距离很小时则直接使其静止不动。
- 第26-59行为判断篮球是否进入篮筐，其判断方法是，查看其篮球中心和篮筐中心的距离是否小于某一个值，而如果下一刻该球在篮筐中心的正下方一点，则说明篮球进入了篮筐，当篮球就入篮筐后，会新建一个线程来使篮网左右扰动一下。
- 第60-79行为判断篮球是否和篮板、墙及地板碰撞，如果碰撞了则播放碰撞声音。

（12）最后开发的是该类中的产生纹理id的方法initTexture，其详细情况如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的GLGameView.java。

```
public int initTexture(Bitmap bitmapTmp,boolean needRrelease){//生成纹理ID
        int[] textures = new int[1];
        GLES20.glGenTextures(
                1,                                          //产生的纹理id的数量
                textures,                                   //纹理id的数组
                0                                           //偏移量
        );
        int textureId=textures[0];                          //存放纹理id的数组
        GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureId);
            GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
                GLES20.GL_TEXTURE_MIN_FILTER,GLES20.GL_NEAREST);
            GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
                GLES20.GL_TEXTURE_MAG_FILTER,GLES20.GL_LINEAR);
            GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
                 GLES20.GL_TEXTURE_WRAP_S,GLES20.GL_CLAMP_TO_EDGE);
            GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
                GLES20.GL_TEXTURE_WRAP_T,GLES20.GL_CLAMP_TO_EDGE);
```

```
          GLUtils.texImage2D (
          GLES20.GL_TEXTURE_2D,                      //纹理类型
           0,
           GLUtils.getInternalFormat(bitmapTmp),
           bitmapTmp,                                //纹理图像
           GLUtils.getType(bitmapTmp),
           0 );                                      //纹理边框尺寸
         if(needRrelease) {
           bitmapTmp.recycle();                      //纹理加载成功后释放图片
         }
         return textureId;
        }
```

说明

该方法为根据图片生成一个纹理id，在绘制需要纹理的物体时，将纹理id传入即可，而该类中通过io加载图片和初始化图像的方法此处就不再对其进行详细介绍了，请读者自行查阅随书光盘中的源代码。

21.6.2　着色器程序管理器类ShaderManager

该类将所有着色器程序管理起来，包括加载着色器程序和生成着色器程序均由该类完成，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的ShaderUtil.java。

```
package com.bn.tl;
import android.content.res.Resources;                          //导入相关包
public class ShaderManager{
    final static int shaderCount=5;                            //shader 程序数
    final static String[][] shaderName={                       //脚本文件名称数组
        {"vertex.sh","frag.sh"},                               //只有纹理的 shader 程序
        {"vertex_yingzi.sh","frag_yingzi.sh"},                 //有光照影子的 shader 程序
        {"lightvertex.sh","lightfrag.sh"},                     //只有光照的 shader 程序
        {"vertex.sh","frag_blackground.sh"},                   //仪表板背景的 shader 程序
            {"vertex_net.sh","frag_net.sh"}                    //篮网的 shader 程序
        };
        static String[]mVertexShader=new String[shaderCount];
                                                               //存放顶点着色器脚本字符串的数组
        static String[]mFragmentShader=new String[shaderCount];
                                                               //存放片元着色器脚本字符串的数组
        static int[] program=new int[shaderCount];             //存放程序 shader 程序 id 的数组
        public static void loadCodeFromFile(Resources r){      //加载脚本成字符串方法
          for(int i=0;i<shaderCount;i++){
            mVertexShader[i]=ShaderUtil.loadFromAssetsFile(shaderName[i][0],r);
            mFragmentShader[i]=ShaderUtil.loadFromAssetsFile(shaderName[i][1], r);
          }}
        public static void compileShader(){              //这里主要是编译 3D 中欢迎界面的着色器
            for(int i=0;i<1;i++){
                program[i]=ShaderUtil.createProgram(mVertexShader[i], mFragment
                Shader[i]);
            }}
        public static void compileShaderReal(){          //编译生成 3D 物体的 shader 程序方法
            for(int i=1;i<shaderCount;i++){
                program[i]=ShaderUtil.createProgram(mVertexShader[i], mFragment
                Shader[i]);
            }}
        public static int getCommTextureShaderProgram(){//这里返回的是普通的 shader 程序
            return program[0];
        }
        public static int getShadowshaderProgram(){ //这里返回的是影子的 shader 程序
            return program[1];
        }
        public static int getLigntAndTexturehaderProgram(){
                                                    //这里返回的是光照纹理的 shader 程序
            return program[2];
        }
        public static int getBlackgroundShaderProgram(){
                                                    //这里返回的是仪表板背景的shader 程序
            return program[3];
        }
        public static int getBasketNetShaderProgram(){  //这里返回的是篮网的 shader 程序
            return program[4];
        }}
```

说明

该类为shader程序管理类，该类提供加载shader脚本字符串方法和链接生成对应shader程序方法。然后提供返回各种功能的shader程序方法，在使用时直接调用对应方法得到对应功能程序即可。

21.6.3 添加刚体进物理世界类TianjiaBody

该类只提供了一个构造器，每次创建该类对象时，只是完成了将对应物体的刚体添加进物理世界，以供其模拟计算。该类的详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的TianjiaBody.java。

```
1 package com.bn.tl;
2 ……//此处省略了部分类和包的引入代码，读者可自行查阅光盘的源代码
3 public class TianjiaBody{
4     RigidBody gangti;
5     public int isnoLanBan=0;                                   //是否和篮板在前一刻碰撞了
6     public TianjiaBody(CollisionShape colShape,                //物体形状
7         DiscreteDynamicsWorld dynamicsWorld,float mass,        //物理世界
8         float cx,float cy,float cz,float restitution,float friction){    //对应位置
9         Transform groundTransform = new Transform();           //创建刚体的初始变换对象
10            groundTransform.setIdentity();
11            groundTransform.origin.set(new Vector3f(cx, cy, cz));          //设置位置
12            Vector3f localInertia = new Vector3f(0, 0, 0);     //惯性设置为零，即静止物体
13            DefaultMotionState myMotionState =                 //创建刚体的运动状态对象
14                    new DefaultMotionState(groundTransform);
15            RigidBodyConstructionInfo rbInfo = new             //创建刚体信息对象
16                    RigidBodyConstructionInfo(0, myMotionState, colShape,
              localInertia);
17            gangti = new RigidBody(rbInfo);                    //创建刚体
18            gangti.setRestitution(restitution);                //设置反弹系数
19            gangti.setFriction(1f);                            //设置摩擦系数
20            dynamicsWorld.addRigidBody(gangti,                 //将刚体添加进物理世界
21                    Constant.GROUP_HOUSE, Constant.MASK_HOUSE);
22        }}
```

- 第9-14行为设置该物体的位置和运动状态，以及其是否为静止物体的属性。此处没有对刚体的密度进行设置，则说明此添加进来的刚体为静止不动的。

- 第15-21行为根据物体形状创建对应的刚体，最后将刚体添加进物理世界。

21.6.4　添加胶囊刚体进物理世界类JiaoNangTianjiaBody

该类也只提供了一个构造器，将胶囊刚体添加进物理世界，设置刚体的位置和摆放状态，即包括设置其在3D世界中的位置和旋转情况。其作用是用胶囊来搭建一个篮筐。该类的详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的JiaoNangTianjiaBody.java。

```java
package com.bn.tl;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘的源代码
public class JiaoNangTianjiaBody{
    float yAngle;
    RigidBody gangti;
    public int isnoLanBan=0;                //是否和篮板在前一刻碰撞了，0表示没有碰撞
    public JiaoNangTianjiaBody(CollisionShape colShape,
            DiscreteDynamicsWorld dynamicsWorld,float mass,
    float cx,float cy,float cz,float restitution,float friction,float xAngle,float
    yAngle,float zAngle){
            this.yAngle=yAngle;
            boolean isDynamic = (mass != 0f);                      //查看是否设置密度
            Vector3f localInertia = new Vector3f(0, 0, ZJ_LENGTH/2); //中心坐标
            if (isDynamic) {
                colShape.calculateLocalInertia(mass, localInertia); //设置密度
            }
            Transform groundTransform = new Transform();     //创建刚体的初始变换对象
            groundTransform.setIdentity();
            groundTransform.origin.set(new Vector3f(cx, cy, cz));    //设置位置
            groundTransform.basis.rotX((float)Math.toRadians(xAngle));//绕X轴旋转
            groundTransform.basis.rotY((float)Math.toRadians(yAngle));//绕Y轴旋转
            groundTransform.basis.rotZ((float)Math.toRadians(zAngle)); //绕Z轴旋转
            DefaultMotionState myMotionState=new DefaultMotionState(groundTransform);
            RigidBodyConstructionInfo rbInfo = new RigidBodyConstructionInfo(
                mass, myMotionState, colShape, localInertia     //创建刚体信息对象
            );
            gangti = new RigidBody(rbInfo);                        //创建刚体
            gangti.setRestitution(restitution);                    //设置反弹系数
            gangti.setFriction(1f);                                //设置摩擦系数
            dynamicsWorld.addRigidBody(gangti,                     //将刚体添加进物理世界
            Constant.GROUP_HOUSE, Constant.MASK_HOUSE);
    }
```

说明

该类和添加刚体类没有太大的区别，该类多了一个对刚体旋转信息的设置创建胶囊时的初始朝向是*Z*轴，要使其构建篮筐这样的圆环，则需要对其进行对应的旋转。还多了一个对刚体密度的设置，当密度不为零时表示该刚体不为静止不动物体。详细情况请见注释。

21.6.5　篮球控制类BasketBallForDraw

篮球控制类，根据该篮球对应刚体的运动状态，在3D世界中绘制对应的篮球。从对应刚体中获取位置坐标和旋转情况，最后绘制出来，该类的详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/t1目录下的BasketBallForDraw.java。

```
package com.bn.tl;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘的源代码
public class BasketBallForDraw {
    BasketBallTextureByVertex ball;                                          //球的引用
    RigidBody body;                                                          //对应的刚体对象
    float cx,cy,cz;                                                          //篮球所在位置
    int ball_State=0;            //1 表示处于篮筐上部且向下运动.其他情况下为 0
    public  int isnoLanBan=0;    //是否和篮板在前一刻碰撞了，0 表示没有碰撞
    public  int isnoLanQuan=0;   //是否和篮圈在前一刻碰撞了，1 表示碰撞了
        public BasketBallForDraw  (                                          //该类的构造器
            BasketBallTextureByVertex ball,CollisionShape colShape,
            DiscreteDynamicsWorld dynamicsWorld,float mass,
            float cx,float cy,float cz,short group,short mask)  {
            this.cx=cx;this.cy=cy;this.cz=cz;                                //位置
            this.ball=ball;
            boolean isDynamic = (mass != 0f);                                //查看是否设置密度
            Vector3f localInertia = new Vector3f(0, 0, 0);                   //中心坐标
            if (isDynamic)    {
                colShape.calculateLocalInertia(mass, localInertia);  //设置密度
            }
            Transform startTransform = new Transform();      //创建刚体的初始变换对象
            startTransform.setIdentity();
            startTransform.origin.set(new Vector3f(cx, cy, cz));       //位置
            DefaultMotionState myMotionState = new DefaultMotionState(startTransform);
            RigidBodyConstructionInfo rbInfo = new RigidBodyConstructionInfo(
                mass, myMotionState, colShape, localInertia);   //创建刚体信息对象
            body = new RigidBody(rbInfo);                                    //创建刚体
            body.setRestitution(0.4f);                                       //设置反弹系数
            body.setFriction(0.8f);                                          //设置摩擦系数
            body.setActivationState(CollisionObject.WANTS_DEACTIVATION); //首次是静止的
            dynamicsWorld.addRigidBody(body,group,mask);                     //添加进物理世界
        }
        public void drawSelf(int ballTexId,int isShadow,int planeId,int isLanbanYy){
            try{
                Transform trans = body.getMotionState().getWorldTransform(new
```

```
                Transform());
37              Quat4f ro=trans.getRotation(new Quat4f());  //获取篮球的变换信息对象
38             MatrixState.pushMatrix();                     //保护现场
39              cx=trans.origin.x;                           //得到其位置坐标
40              cy=trans.origin.y;
41              cz=trans.origin.z;
42              if(isShadow==1){                             //如果是绘制影子
43                  GLES20.glEnable(GLES20.GL_DEPTH_TEST);//开启深度检测
44              }
45              else if(cz>3*Constant.QIU_R&&cy<Constant.QIU_R*2.5f){
                                                             //如果篮球在一定范围内
46                  GLES20.glDisable(GLES20.GL_DEPTH_TEST);//关闭深度检测
47              }else    {
48                  GLES20.glEnable(GLES20.GL_DEPTH_TEST); //否则开启深度检测
49              }
50            MatrixState.translate(cx,cy,cz);               //推动到对应位置
51              if(ro.x!=0||ro.y!=0||ro.z!=0)  {
52                  float[] fa=SYSUtil.fromSYStoAXYZ(ro);  //得到其旋转信息
53                  MatrixState.rotate(fa[0], fa[1], fa[2], fa[3]); //绕对应轴旋转
54              }
55              ball.drawSelf(ballTexId, isShadow,planeId,isLanbanYy);//绘制
56              MatrixState.popMatrix();                     //恢复现场
57          }
58          catch(Exception e){                              //异常处理
59              e.printStackTrace();
60          }}}
```

• 第10-32行为该类的构造器，其和添加胶囊刚体类的构造器没有太大的区别，都是先初始化刚体的对应值，然后添加进物理世界。

• 第33-60行为根据篮球对应的刚体在物理世界中的变换信息来绘制对应的篮球。其中第36-37行为得到该刚体的变换信息对象，第42-49行为根据其当前绘制的是影子和篮球所在位置来确定是否要开启深度检测。最后根据刚体的位置信息和旋转信息绘制篮球。

21.6.6 纹理矩形绘制类WenLiJuXing

该类为构造一个纹理矩形类，传入纹理矩形的宽度和高度，使用顶点坐标来构造一个矩形，然后绘制出来，该类的详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/src/com/bn/tl目录下的WenLiJuXing.java。

```
package com.bn.tl;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘的源代码
public class WenLiJuXing {
    int mProgram;                                              //自定义渲染管线着色器程序id
   int muMVPMatrixHandle;                                      //总变换矩阵引用
   int maPositionHandle;                                       //顶点位置属性引用
   int maTexCoorHandle;                                        //顶点纹理坐标属性引用
   private FloatBuffer  mVertexBuffer;                         //顶点坐标数据缓冲
   private FloatBuffer  mTextureBuffer;                        //顶点着色数据缓冲
```

```
int vCount;                                          //顶点数量
int texId;                                           //纹理 Id
public WenLiJuXing(float width,float height,Resources r,float[] textures){
    vCount=6;                                        //顶点个数
    float vertices[]={                               //构造顶点坐标数组
            -width/2,0,-height/2,
            -width/2,0,height/2,
            width/2,0,height/2,
            -width/2,0,-height/2,
            width/2,0,height/2,
            width/2,0,-height/2};
    ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
                                                     //创建顶点坐标数据缓冲
    vbb.order(ByteOrder.nativeOrder());              //设置字节顺序为本地操作系统顺序
    mVertexBuffer = vbb.asFloatBuffer();             //转换为 Float 型缓冲
    mVertexBuffer.put(vertices);                     //向缓冲区中放入顶点坐标数据
    mVertexBuffer.position(0);                       //设置缓冲区起始位置
    ByteBuffer tbb = ByteBuffer.allocateDirect(textures.length*4);
                                                     //创建顶点纹理数据缓冲
    tbb.order(ByteOrder.nativeOrder());              //设置字节顺序为本地操作系统顺序
    mTextureBuffer= tbb.asFloatBuffer();             //转换为 Float 型缓冲
    mTextureBuffer.put(textures);                    //向缓冲区中放入顶点着色数据
    mTextureBuffer.position(0);                      //设置缓冲区起始位置
}
 public WenLiJuXing(float width,float height){
    vCount=6;                                        //顶点个数
    float vertices[]= {                              //位置坐标构造矩形
            -width/2,0,-height/2,
            -width/2,0,height/2,
            width/2,0,height/2,
            -width/2,0,-height/2,
            width/2,0,height/2,
            width/2,0,-height/2};
    ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
                                                     //创建顶点坐标数据缓冲
    vbb.order(ByteOrder.nativeOrder());              //设置字节顺序为本地操作系统顺序
    mVertexBuffer = vbb.asFloatBuffer();             //转换为 int 型缓冲
    mVertexBuffer.put(vertices);                     //向缓冲区中放入顶点坐标数据
    mVertexBuffer.position(0);                       //设置缓冲区起始位置
    float textures[]={0,0,0,1f,1f,1f, 0,0,1f,1f,1f,0};  //对应顶点的纹理坐标数组
    ByteBuffer tbb = ByteBuffer.allocateDirect(textures.length*4);
                                                     //创建顶点纹理数据缓冲
    tbb.order(ByteOrder.nativeOrder());              //设置字节顺序为本地操作系统顺序
    mTextureBuffer= tbb.asFloatBuffer();             //转换为 Float 型缓冲
    mTextureBuffer.put(textures);                    //向缓冲区中放入顶点着色数据
    mTextureBuffer.position(0);                      //设置缓冲区起始位置
 }
public void intShader(int mProgram) {                //初始化着色器的 intShader 方法
    this.mProgram=mProgram;                          //程序 ID
    maPositionHandle = GLES20.glGetAttribLocation//获取程序中顶点位置属性引用
```

```
56                (mProgram, "aPosition");
57           maTexCoorHandle= GLES20.glGetAttribLocation//获取程序中顶点纹理坐标属性引用
58                (mProgram, "aTexCoor");
59           muMVPMatrixHandle = GLES20.glGetUniformLocation    //获取程序中总变换矩阵引用
60                (mProgram, "uMVPMatrix");
61       }
62       public void drawSelf(int texId){
63           GLES20.glUseProgram(mProgram);                //制定使用某套着色器程序
64           GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, //将最终变换矩阵传入着色器程序
65                false, MatrixState.getFinalMatrix(), 0);
66           GLES20.glVertexAttribPointer (                      //传入顶点位置数据
67                maPositionHandle,3, GLES20.GL_FLOAT,
68                false, 3*4, mVertexBuffer);
69           GLES20.glVertexAttribPointer(                       //传入顶点纹理坐标数据
70                maTexCoorHandle, 2,GLES20.GL_FLOAT,
71                false, 2*4, mTextureBuffer);
72           GLES20.glEnableVertexAttribArray(maPositionHandle);  //允许顶点位置数据数组
73           GLES20.glEnableVertexAttribArray(maTexCoorHandle);   //允许纹理坐标数据
74           GLES20.glActiveTexture(GLES20.GL_TEXTURE0);          //绑定纹理
75           GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, texId);
76           GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount);  //绘制纹理矩形
77       }}
```

- 第12-31行为该类的一个构造器，传入矩形的长、宽和纹理坐标数组构造一个矩形，然后生成对应的顶点缓冲数据和纹理坐标缓冲数据。
- 第32-52行为该类的另外一个构造器，此构造器只需传入长和宽，即可生成对应的顶点缓冲数据和纹理坐标缓冲数据。
- 第53-61行为初始化shader程序，即从对应的着色器程序中获取着色器中对应变量属性id。
- 第62-77行为绘制方法，根据对应的属性id传入对应的数据，首先设置其使用某套shader程序，然后将变换矩阵、顶点位置数据、纹理坐标数据传入对应的属性id，然后打开使用顶点位置数据和纹理坐标数据，最后绑定纹理进行绘制纹理矩形。

说明

该游戏中对物体的具体绘制的方法大致相同，所以在这里只为读者介绍纹理矩形的绘制类，对其他物件的具体绘制在本章中就不为大家一一介绍了，请读者自行查阅随书光盘中的源代码。

20.7　游戏中着色器的开发

前一小节对绘制游戏界面的相关类进行了介绍，这一小节将对该游戏中用到的相关着色器进行介绍，本游戏中用到的着色器有纹理的着色器、光照的着色器、影子着色器、仪表板背景着色器和篮网着色器。下面就对该游戏中用到的着色器的开发一一进行介绍。

21.7.1 纹理的着色器

纹理着色器分为顶点着色器和片元着色器，下面便分别对纹理着色器的顶点着色器和片元着色 器的开发进行介绍。

（1）首先开发的是纹理着色器的顶点着色器，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/assets目录下的vertex.sh。

```
uniform mat4 uMVPMatrix;                                  //总变换矩阵
attribute vec3 aPosition;                                 //顶点位置
attribute vec2 aTexCoor;                                  //顶点纹理坐标
varying vec2 vTextureCoord;                               //用于传递给片元着色器的变量
void main(){
   gl_Position = uMVPMatrix * vec4(aPosition,1);          //根据总变换矩阵计算此顶点位置
   vTextureCoord = aTexCoor;                              //将接收的纹理坐标传递给片元着色器
}
```

> **说明**
>
> 该顶点着色器的作用主要为根据顶点位置和总变换矩阵计算gl_Position，每顶点执行一次。

（2）完成顶点着色器的开发后，下面开发的是纹理着色器的片元着色器，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/assets目录下的frag.sh。

```
precision mediump float;
varying vec2 vTextureCoord;                                //接收从顶点着色器过来的参数
uniform sampler2D sTexture;                                //纹理内容数据
void main() {
  vec4 cTemp=texture2D(sTexture, vTextureCoord);           //给此片元从纹理中采样出颜色值
  gl_FragColor = cTemp;                                    //将颜色值赋值给最终颜色
}
```

说明

该片元着色器的作用主要为，根据从顶点着色器传递过来的参数vTextureCoord和从Java代码部分传递过来的sTexture计算片元的最终颜色值，每片元执行一次。

21.7.2　光照着色器

光照着色器为通过光源位置和当前物体位置，来计算物体表面的明暗程度，该光照着色器为纯色物体的光照，用于绘制篮圈。

（1）首先开发的是光照着色器的顶点着色器，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/assets目录下的lightvertex.sh。

```
uniform mat4 uMVPMatrix;                 //总变换矩阵
uniform mat4 uMMatrix;                   //变换矩阵
uniform vec3 uLightLocation;             //光源位置
uniform vec3 uCamera;                    //摄像机位置
attribute vec3 aPosition;                //顶点位置
attribute vec3 aNormal;                  //顶点法向量
varying vec4 vambient;                   //用于传递给片元着色器的环境光变量
varying vec4 vdiffuse;                   //用于传递给片元着色器的散射光变量
```

```
9 varying vec4 vspecular;                          //用于传递给片元着色器的镜面反射光变量
10    void pointLight(                             //定位光光照计算的方法
11      in vec3 normal,                            //法向量
12      inout vec4 vambient,                       //环境光最终强度
13      inout vec4 vdiffuse,                       //散射光最终强度
14      inout vec4 vspecular,                      //镜面光最终强度
15      in vec3 ulightLocation,                    //光源位置
16      in vec4 lightAmbient,                      //环境光强度
17      in vec4 lightDiffuse,                      //散射光强度
18      in vec4 lightSpecular                      //镜面光强度
19    ){
20      vambient=lightAmbient;                     //直接得出环境光的最终强度
21      vec3 normalTarget=aPosition+normal;        //计算变换后的法向量
22      vec3 newNormal=(uMMatrix*vec4(normalTarget,1)).xyz-(uMMatrix*vec4
     (aPosition,1)).xyz;
23      newNormal=normalize(newNormal);            //对法向量规格化
24      //计算从表面点到摄像机的向量
25      vec3 eye= normalize(uCamera-(uMMatrix*vec4(aPosition,1)).xyz);
26      //计算从表面点到光源位置的向量 vp
27      vec3 vp= normalize(ulightLocation-(uMMatrix*vec4(aPosition,1)).xyz);
28      vp=normalize(vp);                                  //格式化 vp
29      vec3 halfVector=normalize(vp+eye);                 //求视线与光线的半向量
30      float shininess=50.0;                              //粗糙度，越小越光滑
31      float nDotViewPosition=max(0.0,dot(newNormal,vp));//求法向量与 vp 的点积与 0 的最大值
32      vdiffuse=lightDiffuse*nDotViewPosition;            //计算散射光的最终强度
33      float nDotViewHalfVector=dot(newNormal,halfVector);//法线与半向量的点积
34      float powerFactor=max(0.0,pow(nDotViewHalfVector,shininess)); //镜面反射光强度因子
35      vspecular=lightSpecular*powerFactor;               //计算镜面光的最终强度
36    }
37    void main(){
38       gl_Position = uMVPMatrix * vec4(aPosition,1); //根据总变换矩阵计算此次绘制此顶点位置
39       vec4 ambientTemp=vec4(0.0,0.0,0.0,0.0);           //初始化环境光、散射光和镜反射光
40       vec4 diffuseTemp=vec4(0.0,0.0,0.0,0.0);
41       vec4 specularTemp=vec4(0.0,0.0,0.0,0.0);
42       pointLight(normalize(aNormal),ambientTemp,diffuseTemp,specularTemp,
         uLightLocation,
43              vec4(0.1,0.1,0.1,1.0),vec4(0.7,0.7,0.7,1.0),vec4(0.3,0.3,0.3,1.0));
44       vambient=ambientTemp;                             //将环境光赋值
45       vdiffuse=diffuseTemp;                             //将散射光赋值
46       vspecular=specularTemp;                           //将镜面反射光赋值
47    }
```

- 第10-36行为计算3种光，其根据光源位置、顶点位置和该点法向量来计算光照强度，实质为计算其对应点的颜色值，计算光照的公式此处就不再对其进行详细介绍了。
- 第37-47行为主方法，首先计算出点的位置，最后使用计算光照方法计算出3种光的颜色，最后将3种光的颜色值传入片元着色器。

（2）完成顶点着色器的开发后，下面开发的是片元着色器，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/assets目录下的lightfrag.sh。

```
precision mediump float;
varying vec4 vambient;                       //接收从顶点着色器过来的参数
varying vec4 vdiffuse;                       //环境光、散射光和镜面反射光
varying vec4 vspecular;
void main(){                                 //将计算出的颜色给此片元
   vec4 finalColor=vec4(0.9,0.01,0.01,1.0);  //物体颜色
   gl_FragColor = finalColor*ambient+finalColor*specular+finalColor*diffuse;
                                             //给此片元颜色值
}
```

说明

其原理为在顶点着色器处根据其位置计算得到的相关光照强度，然后根据对应光的强度和自身颜色得出最终该片元的颜色。

21.7.3　影子着色器

影子着色器为根据光源位置和物体位置，计算其在某个平面上的影子，其原理为将物体上所有的点通过光源投影到指定平面上，然后将所有点的颜色指定为影子颜色即可。

（1）首先开发的是影子着色器的顶点着色器。其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/assets目录下的vertex_yingzi.sh。

```
uniform int uisShadow;                          //是否为绘制影子
uniform mat4 uMVPMatrix;                        //总变换矩阵
uniform mat4 uMMatrix;                          //变换矩阵
uniform mat4 uMCameraMatrix;                    //摄像机矩阵
uniform mat4 uMProjMatrix;                      //投影矩阵
uniform vec3 uLightLocation;                    //光源位置
uniform vec3 uCamera;                           //摄像机位置
uniform vec3 uplaneN;                           //需要影子平面的法向量
uniform vec3 uplaneA;                           //需要影子平面的一个点
    attribute vec3 aPosition;                   //顶点位置
    attribute vec3 aNormal;                     //顶点法向量
    attribute vec2 aTexCoor;                    //顶点纹理坐标
    varying vec2 vTextureCoord;                 //用于传递给片元着色器的变量
    varying vec4 vambient;                      //环境光
    varying vec4 vdiffuse;                      //散射光
    varying vec4 vspecular;                     //镜面反射光
    varying vec4 vfragLosition;                 //顶点位置
    void pointLight(                            //定位光光照计算的方法
      in vec3 normal,                           //法向量
      inout vec4 vambient,                      //环境光最终强度
      inout vec4 vdiffuse,                      //散射光最终强度
      inout vec4 vspecular,                     //镜面光最终强度
      in vec3 ulightLocation,                   //光源位置
      in vec4 lightAmbient,                     //环境光强度
      in vec4 lightDiffuse,                     //散射光强度
```

```
26    in vec4 lightSpecular                              //镜面光强度
27  ){
28    vambient=lightAmbient;                             //直接得出环境光的最终强度
29    vec3 normalTarget=aPosition+normal;                //计算变换后的法向量
30    vec3 newNormal=(uMMatrix*vec4(normalTarget,1)).xyz-(uMMatrix*vec4(aPosition,1)).xyz;
31    newNormal=normalize(newNormal);                    //对法向量规格化
32    //计算从表面点到摄像机的向量
33    vec3 eye= normalize(uCamera-(uMMatrix*vec4(aPosition,1)).xyz);
34    //计算从表面点到光源位置的向量 vp
35    vec3 vp= normalize(ulightLocation-(uMMatrix*vec4(aPosition,1)).xyz);
36    vp=normalize(vp);                                  //格式化 vp
37    vec3 halfVector=normalize(vp+eye);                 //求视线与光线的半向量
38    float shininess=50.0;                              //粗糙度，越小越光滑
39    float nDotViewPosition=max(0.0,dot(newNormal,vp));//求法向量与 vp 的点积与 0 的最大值
40    vdiffuse=lightDiffuse*nDotViewPosition;            //计算散射光的最终强度
41    float nDotViewHalfVector=dot(newNormal,halfVector);    //法线与半向量的点积
42    float powerFactor=max(0.0,pow(nDotViewHalfVector,shininess)); //镜面反射光强度因子
43    vspecular=lightSpecular*powerFactor;               //计算镜面光的最终强度
44  }
45  void main() {
46     if(iusShadow==1) {                                //若为阴影则根据公式进行投影计算
47       vec3 A=uplaneA;                                 //投影面的一个点
48       vec3 n=uplaneN;                                 //投影平面的法向量
49       vec3 S=uLightLocation;                          //光源位置
50       vec3 V=(uMMatrix*vec4(aPosition,1)).xyz;        //计算变换后的坐标
51       vec3 VL=S+(V-S)*(dot(n,(A-S))/dot(n,(V-S)));
52   fragLosition= uMProjMatrix*uMCameraMatrix*vec4(VL,1); //根据总变换矩阵计算此顶点位置
53       gl_Position=fragLosition;                       //赋值给最终点位置
54       vec4 ambientTemp=vec4(0.0,0.0,0.0,0.0);         //初始化各种光照
55       vec4 diffuseTemp=vec4(0.0,0.0,0.0,0.0);
56       vec4 specularTemp=vec4(0.0,0.0,0.0,0.0);
57       pointLight(normalize(aNormal),ambientTemp,diffuseTemp,   //计算光照
58  specularTemp,uLightLocation,vec4(0.3,0.3,0.3,0.3),vec4(0.7,0.7,0.7,0.3),vec4
    (0.3,0.3,0.3,0.3));
59       vambient=ambientTemp;                           //最后传入片元着色器
60       vdiffuse=diffuseTemp;
61       vspecular=specularTemp;
62     } else {
63         vfragLosition= uMVPMatrix * vec4(aPosition,1);//根据总变换矩阵计算此顶点位置
64         gl_Position =vfragLosition;
65         vec4 ambientTemp=vec4(0.0,0.0,0.0,0.0);
66         vec4 diffuseTemp=vec4(0.0,0.0,0.0,0.0);
67         vec4 specularTemp=vec4(0.0,0.0,0.0,0.0);
68         pointLight(normalize(aNormal),ambientTemp,diffuseTemp,//计算光照
69  specularTemp,uLightLocation,vec4(0.3,0.3,0.3,1.0),vec4(0.7,0.7,0.7,1.0),vec4
    (0.3,0.3,0.3,1.0));
70         vambient=ambientTemp;                         //最后传入片元着色器
71         vdiffuse=diffuseTemp;
72         vspecular=specularTemp;
73       }

74     vTextureCoord = aTexCoor;                     //将接收的纹理坐标传递给片元着色器
75  }
```

- 第18-44行为定位光的计算方法，传入光源位置、点坐标和点的法向量，即可计算出对应3种光的强度，在绘制篮球的时候用到光

照。

- 第45-75行为主方法，其中第46～61行为绘制篮球的影子，将该点根据光源和要绘制影子的平面，计算出该点投影到影子平面的位置点，即为最终影子的一个点。
- 第62-75行为绘制带有光照的篮球，同样根据光源位置和自身位置，计算光照强度，最后传入片元着色器。第*74* 行为将接收到的纹理坐标传给片元着色器。

（2）完成顶点着色器的开发后，下面开发的是影子着色器的片元着色器，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/assets目录下的frag_yingzi.sh。

```
precision mediump float;
uniform int uisShadowFrag;                                    //是否绘制阴影
uniform int uisLanbanFrag;                                    //是否绘制球在篮板上的阴影
uniform sampler2D usTextureBall;                              //平面纹理内容数据
varying vec2 vTextureCoord;                                   //接收从顶点着色器过来的参数
varying vec4 vambient;                                        //对应光照
varying vec4 vdiffuse;
varying vec4 vspecular;
varying vec4 vfragLosition;                                   //从顶点着色器传过来的顶点坐标
  void main(){
    vec4 finalColor;
    if(uisShadowFrag==1) {                                    //绘制阴影，纹理从桌面纹理采样
     finalColor=vec4(0.3,0.3,0.3,0.2);                        //影子颜色
      if(uisLanbanFrag==1)  {                                 //如果是在篮板上绘制影子
       if(vfragLosition.x<-9.6||vfragLosition.x>9.6|| //超过其范围则完全使其透明
           vfragLosition.y<4.24||vfragLosition.y>10.5){
        finalColor=vec4(0.3,0.3,0.3,0.0);
      } }
      gl_FragColor = finalColor;                                //给此片元颜色值
    } else {                                                    //绘制球本身
      finalColor=texture2D(usTextureBall, vTextureCoord);//纹理从球纹理采样
      gl_FragColor = finalColor*vambient+finalColor*vspecular+finalColor*vdiffuse;
    }}
```

> **说明**
>
> 该着色器分为3种情况，第一种是只绘制篮球；第二种是绘制篮球在篮板上的影子，此时需要在篮板范围以外的地方使其完全透明，才能只在篮板上有影子；第三种是给篮板以外的平面绘制影子。

21.7.4 仪表盘背景着色器

该着色器根据采样出来的纹理颜色，判断如果背景为纯黑色则要将其完全透明，而其顶点着色器和纹理着色器的顶点着色器为同一套，此处对于该顶点着色器将不再重复介绍。下面直接介绍该着色器的片元着色器，其详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/assets目录下的frag_blackground.sh。

```
precision mediump float;
varying vec2 vTextureCoord;                                  //接收从顶点着色器过来的参数
uniform sampler2D sTexture;                                  //纹理内容数据
void main(){
  vec4 cTemp=texture2D(sTexture, vTextureCoord);             //给此片元从纹理中采样出颜色值
  if(cTemp.r==0.0&&cTemp.g==0.0&&cTemp.b==0.0||cTemp.a==0.0) {//如果该片元颜色为纯黑色
     gl_FragColor=vec4(0.3,0.3,0.3,0.0);                     //使其完全透明
  } else {
   gl_FragColor = cTemp;                                     //否则为原来采样出来的颜色
     }}
```

说明

该片元着色器的作用主要为，判断从纹理采样出的颜色值是否为黑色，如果是则将其透明度设置为0，否则以采样的颜色进行绘制。

21.7.5 篮网着色器

篮网着色器为根据传入的扰动值来变动顶点位置，使其可以实时计算顶点的位置，在篮球进球时，根据不断变换扰动值，可实现使篮网左右摇摆的一个动态效果。

该着色器的顶点着色器与其他着色器相比有些许不同，而片元着色器和上面仪表盘背景着色器的片元着色器没有区别，此处就不再对其进行介绍了，顶点着色器的详细代码如下。

代码位置：见随书光盘中源代码/第21章/BasketBall_Shot/assets目录下的vertex_net.sh。

```
1 uniform mat4 uMVPMatrix;                          //总变换矩阵
2 attribute vec3 aPosition;                         //顶点位置
3 uniform int uraodon;                              //扰动值
4 attribute vec2 aTexCoor;                          //顶点纹理坐标
5 varying vec2 vTextureCoord;                       //用于传递给片元着色器的变量
6 void main(){
7    vec4 rPosition=vec4(aPosition,1);              //复制顶点位置
8    float raodonR=0.0;                             //跳动值
9    if(uraodon==1){                                //如果是向右扰动
10         if(rPosition.y<0.46){                    //在小于半个篮网高度时
11           raodonR=sqrt(rPosition.y)/4.0;         //向左扰动按照高度来确定
12           rPosition.x=rPosition.x+raodonR;       //x 方向向右扰动
13         }else{
14           raodonR=sqrt(1.12-rPosition.y)/4.0;    //高于一半以上时，按高度减小扰动值
15           rPosition.x=rPosition.x+raodonR;       //x 方向向右扰动
16         }}
17       else if(uraodon==3) {                      //如果是向左扰动
18         if(rPosition.y<0.46) {                   //同样小于篮网高度一半时
19           raodonR=sqrt(rPosition.y)/4.0;         //根据高度确定扰动值
20           rPosition.x=rPosition.x-raodonR;       //向左边扰动
21         }else {
22           raodonR=sqrt(1.12-rPosition.y)/4.0;    //否则根据高度反方向确定扰动值
23           rPosition.x=rPosition.x-raodonR;       //向左减少其 x 值
24         }}
25         gl_Position =uMVPMatrix * rPosition;     //根据总变换矩阵计算此顶点位置
26         vTextureCoord = aTexCoor;                //将接收的纹理坐标传递给片元着色器
27    }
```

- 第9～16行为当前扰动为向右扰动时，使当前点的x坐标根据当前点所在篮网的高度加上一个扰动值即可。
- 第17～24行为当前扰动为向左扰动，使当前点的x坐标根据当前点所在篮网的高度值减去一个扰动值即可。

21.8　游戏的优化与改进

本章火力篮球游戏基于OpenGL ES 2.0渲染引擎，使用第三方开源物理引擎。虽然本游戏的基本框架已经开发完毕，但是还有着许多地方需要优化与改进。

- 游戏界面的优化。

虽然游戏的整体场景已经很绚丽，但相对于一些大型游戏来说还有一些差距，读者可以根据个人的爱好和习惯做进一步的改进，总体风格可以根据自己喜好重新设计。

- 对游戏玩法的完善。

本游戏只是一个一分钟投篮游戏，只有一个排行榜，如果添加进别的花样可能会使该游戏更具可玩性和挑战性，能让更多的玩家喜欢上此游戏。

- 游戏引擎的完善。

本游戏使用第三方的开源物理引擎，由于该物理引擎使用JBullet，因此运行速度上还不是很理想。如果添加更多的物体进物理世界或者完成更多的功能需求，那么在运行速度上就需要进一步完善了。

第22章

夜鹰行动

随着移动互联网时代的到来，移动手持设备的功能越来越强大。人们在移动设备上进行娱乐的时间越来越多，同时伴随着智能手机模拟现实技术的成熟，各种模拟现实类游戏更是迅猛发展。

本章将开发一款基于Android平台的模拟现实类飞机空战游戏——夜鹰行动，经过这一章的学习，读者将会了解如何结合传感器开发一款飞机空战类3D游戏。

22.1　游戏背景及功能概述

这一节将会对该游戏的背景及其相应功能进行简单介绍，通过本节的学习，读者将会对该游戏的具体功能及相应游戏效果有一个整体的了解。

22.1.1　背景概述

夜鹰行动是一款模拟现实中飞机空战的休闲游戏。通俗意义上讲空战是指利用飞行器在空中进行战斗，以击落敌方，夺取制空权为目的一种战争形式。

第二次世界大战末期，德国首先在飞机上安装了喷气发动机。第二次世界大战结束，冷战开始，在长达五十年的时间里，各国的战斗机发生了很大的变化。喷气发动机的应用改进了飞行特性，机载雷达的应用使飞行的视野大大提高，导弹的出现改变了战斗的方式，电子设备的使用使战斗变得更加扑朔迷离。

夜鹰行动就是一款使用飞行器，以击毁敌方空中目标、陆地军火库、陆地坦克和陆地高射炮为游戏任务的游戏。该游戏分两种模式，即战役模式和特殊行动模式。在战役模式下，须击毁该关地图上所有敌方目标；在特殊行动模式下只须在规定时间内击毁敌方对应目标。

22.1.2　功能介绍

前一小节介绍了游戏背景，这一小节将对该游戏的功能以及操作方法进行简单的介绍，使读者对该游戏有一定的感性认识，具体步骤如下。

（1）运行该游戏，首先进入的是该游戏的视频播放界面，其效果如图22-1和图22-2所示。



图22-1　Logo展示视频（1）



图22-2　Logo展示视频（2 ）

（2）视频播放界面播放完毕后，将进入游戏资源加载界面。在该界面下进行对3D界面中所用到的资源进行加载，此加载界面为3D界面。其效果如图22-3所示。



图22-3　加载界面

（3）系统资源加载完毕后，将进入主菜单界面。在该界面下可以触摸移动旋转导弹来选择对应的功能，其效果如图22-4所示。



图22-4　主菜单界面

（4）主菜单界面下，如果单击设置按钮，则进入设置界面。在设置界面下，可以开启或者关闭游戏中的背景音乐、音乐特效和振动特效，其效果如图22-5所示。



图22-5　设置界面

（5）主菜单界面下，如果单击排行按钮，则进入排行榜界面，排行榜界面的效果如图22-6所示。



图22-6　排行榜界面

（6）主菜单界面下，如果单击帮助按钮，则进入帮助界面，其效果如图22-7所示。



图22-7　帮助界面

（7）主菜单界面下，如果单击关于按钮，则进入的是关于界面，其效果如图22-8所示。



图22-8　关于界面

（8）主菜单界面下，单击开始游戏按钮，进入飞行器选择和游戏模式选择界面，在该界面下，可以通过左右箭头按钮选择飞机，可以通过单击对应的游戏模式选择游戏模式。该游戏有两种游戏模式，一种是战役模式，另一种是特殊行动模式，其效果如图22-9和图22-10所示。



图22-9　飞行器和模式选择界面（1）



图22-10　飞行器和模式选择界面（2）

（9）飞行器选择和游戏模式选择界面下，单击确定按钮时，进入的是对应模式下的关卡选择界面，两种模式对应两个关卡选择界面，其效果如图22-11和图22-12所示。

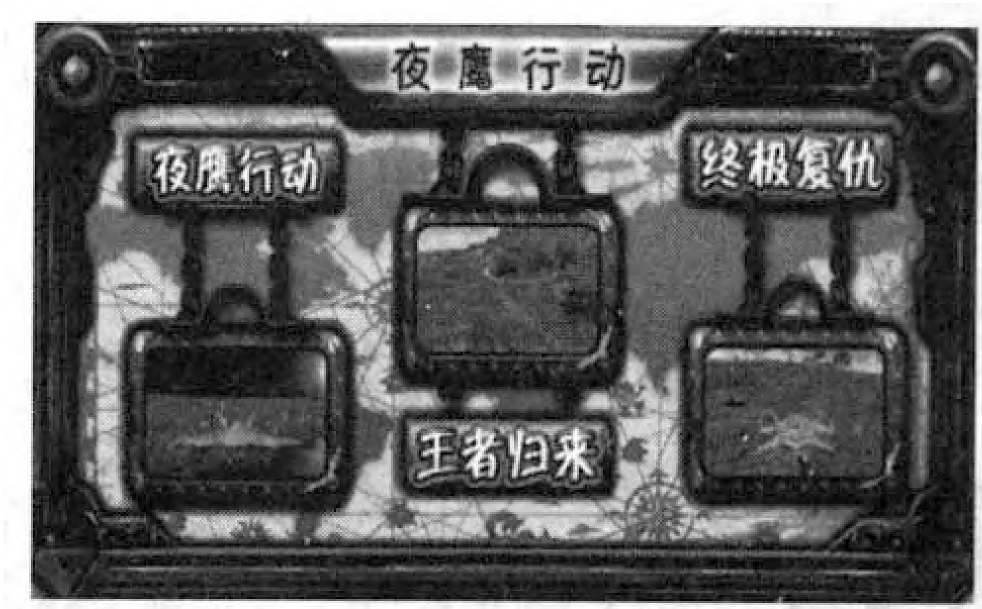


图22-11　战役模式选关界面



图22-12　特殊行动模式选关界面

（10）选关界面下，单击对应的地图，可进入相应的关卡。选关界面下单击一个地图进入游戏时，首先进入游戏场景中的场景预览界面。游戏场景中的场景预览界面下，单击左下角的播放/暂停按钮来播放或者暂停预览，单击右下角的停止预览按钮进入游戏，其效果如图22-13和图22-14所示。



图22-13　战役模式场景预览



图22-14　特殊行动模式场景预览

（11）场景预览界面下，单击右下角的停止预览按钮时，进入游戏场景界面，在该界面下通过改变手机姿态来控制飞机方向，通过左下方的上下按钮来控制飞机的俯仰，通过右下方的发射按钮 发射子弹或者炮弹，游戏场景的效果如图22-15和图22-16所示。



图22-15　战役模式游戏场景

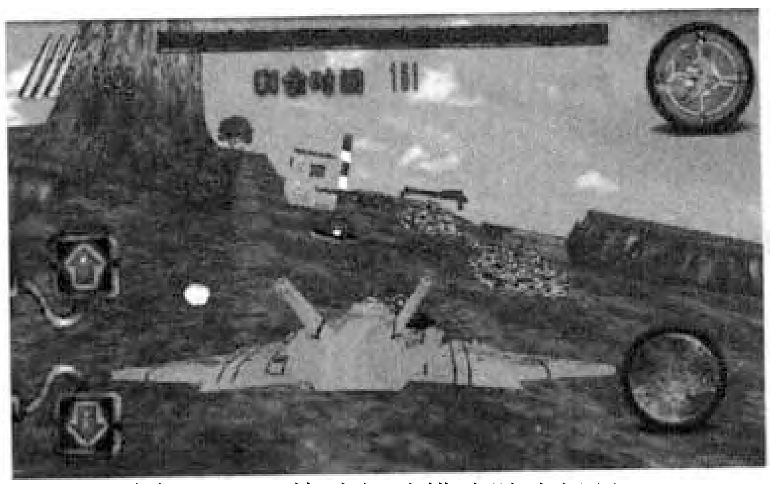


图22-16　特殊行动模式游戏场景

（12）在游戏界面下按下返回按钮或者玩家被击落时，均弹出对话框，此时可选择继续按钮继续游戏，或者选择菜单按钮返回到主菜单。其效果如图22-17所示。

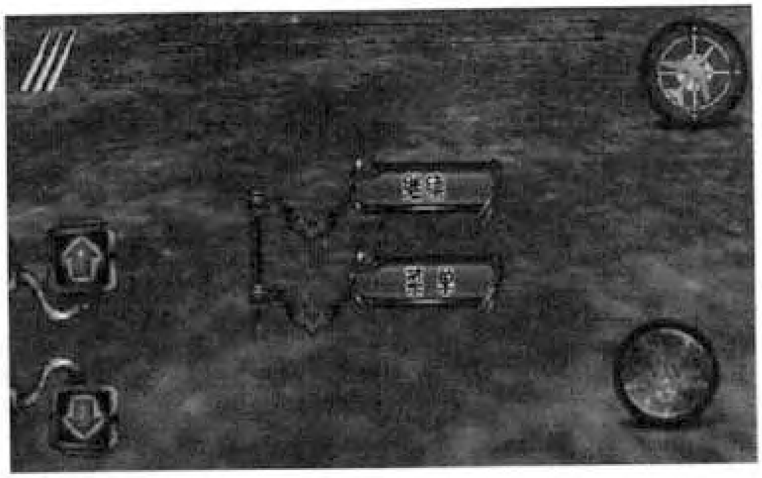


图22-17　游戏场景（1）

（13）在战役模式下当游戏玩家顺利通关时，弹出对话框，此时可选择下一关按钮进入下一关，也可选择重玩按钮重玩此关，还可选择菜单按钮进入选关菜单界面，其效果如图22-18所示。



图22-18　游戏场景（2）

（14）在特殊行动模式下，如果玩家未能完成任务，则弹出行动失败的对话框，其效果如图22-19所示。当玩家顺利通关时弹出行动成功的对话框，其效果如图22-20所示。



图22-19　行动失败对话框



图22-20　行动成功对话框

22.2　游戏的策划及准备工作

完成游戏背景及相应游戏功能的介绍后，这一节将介绍该游戏的整体策划及开发前的一些准备工作。主要包括游戏风格的设计，程序中用到的各种资源的收集和整合。

22.2.1　游戏的策划

首先对游戏的策划进行简单的介绍，真实的游戏项目开发中，游戏的策划会非常的细致、具体、全面。此处由于篇幅所限，仅对该步骤进行了简单的说明，该游戏的策划如下所列。

- 游戏类型。

该游戏使用Android手机中的姿态传感器，以OpenGL ES 2.0作为游戏呈现技术，属于休闲类游戏。

- 运行目标平台。

游戏目标平台为Android 2.2及以上版本，因为使用OpenGL ES 2.0所以必须自带显卡。

- 目标受众。

夜鹰行动操作简单，只需改变手机姿态和控制界面上的几个按钮就能很容易操纵整个游戏。传感器的应用使得整个游戏玩法更具真实感，而OpenGL ES 2.0呈现技术使得游戏场景更加真实逼真，所以适合众多人群。

- 操作方式。

改变手机的姿态来控制飞行器的飞行方向，使用屏幕上的虚拟按钮控制飞机的上下俯仰，用发射按钮控制子弹或者炮弹的发射，而且飞机可自行锁定敌方目标，当锁定一个目标时，自行调整子弹或者炮弹的发射方向，减小了游戏的难度。

- 呈现技术。

本游戏使用OpenGL ES 2.0作为呈现技术，用渐变纹理技术生成的山地具有很强的真实感，游戏模式和不同难度关卡的设定使得游戏更具可玩性。

22.2.2　Android平台下游戏开发的准备工作

完成对游戏策划的介绍后，下面需要做一些开发前的准备工作，包括图片资源的搜集和制作、声音资源的收集和制作等。该游戏用到的资源有，各个物件的贴图图片、游戏中的背景音乐和特性音乐、游

戏欢迎界面视频和游戏场景中用于生成山地的灰度图等，其详细步骤如下。

（1）首先介绍的是本游戏中所用到的图片资源，将用来作为纹理图的图片资源都放在项目文件下的res/drawable-hdpi文件夹下，其详细情况如表22-1所列。

表22-1　图片清单

图片名	大小（KB）	像素（w×h）	用途	图片名	大小（KB）	像素（w×h）	用途
about.jpg	23	64×128	关于按钮	menu_two_left_up.png	9	64×64	向左按钮抬起时
aboutview.png	262	512×1024	关于界面文字图片	menu_two_planeicon_one_down.png	11	64×64	当前选中的飞机是第一架时
action_fail.png	108	256×256	特别行动模式失败时的对话框	menu_two_planeicon_one_up.png	12	64×64	当前选中的飞机不是第一架时
action_plxd.png	72	512×256	霹雳行动说明文字	menu_two_planeicon_three_down.png	11	64×64	当前选中的飞机是第三架时

续表

图片名	大小（KB）	像素（w×h）	用途	图片名	大小（KB）	像素（w×h）	用途
action_smfb.png	74	512×256	沙漠风暴说明文字	menu_two_planeicon_three_up.png	11	64×64	当前选中的飞机不是第三架时
action_win.png	101	256×256	特别行动模式下行动成功时的对话框	menu_two_planeicon_two_down.png	9	64×64	当前选中的飞机是第二架时
action_zsxd.png	70	512×256	斩首行动说明文字	menu_two_planeicon_two_up.png	10	64×64	当前选中的飞机不是第二架时
baozaoxiaoguo.png	100	256×256	飞机坠毁时的爆炸效果	menu_two_right_down.png	8	64×64	向右按钮按下时
baozhazdan.png	82.7	256×256	击毁敌方一个目标时的爆炸效果	menu_two_right_up.png	8	64×64	向右按钮抬起时
barrel_circle_long.png	2.28	64×64	高射炮的炮筒端面贴图	menu_two_war_btn_down.png	17	128×128	当前模式为战役模式时
barrel_circle_short.png	4.14	64×64	高射炮的地面贴图	menu_two_war_btn_up.png	16	128×128	当前模式不为战役模式时
barrel_cylinder_long.png	7.14	64×128	高射炮的炮管贴图	Missile_button.png	29.8	128×128	仪表盘上标记发射炮弹
barrel_cylinder_short.png	2.74	64×32	高射炮的炮台贴图	other.jpg	17.8	64×128	菜单界面下没有选项的格子
bigsmallpingfang.jpg	32.3	256×256	小平房的房顶	music_off.png	45	256×128	打开声音按钮
bigsmallpingfangwuding.jpg	31.9	256×256	小平房的墙	music_on.png	37.8	256×128	关闭声音按钮
bullet_button.png	15.1	128×128	发射子弹标志	nighttexid.jpg	29.8	128×128	灯塔塔身纹理
bullet_purple.png	29.7	128×128	子弹纹理	Number.png	26.6	256×64	数字纹理
button_down.png	13.3	128×128	向下按钮	Missile_cylinder.jpg	5	16×64	菜单导弹身体纹理
button_up.png	23	128×128	向上按钮	Missile_end.jpg	5	32×32	导弹两头纹理
caidanfanhuianniu.png	76	256×256	玩家输时的对话框	pause.png	22	128×128	暂停按钮
caidanshengli.png	108	256×256	玩家赢时的对话框	planebody.jpg	26.6	128×128	机身纹理

续表

图片名	大小（KB）	像素（w×h）	用途	图片名	大小（KB）	像素（w×h）	用途
caodi.jpg	17.1k	128×128	草地贴图	planehead.jpg	29.6	128×128	飞机后机翼纹理
chimney.jpg	14k	64×256	烟囱的纹理图	planehittext.png	12	128×128	飞机被击中时的表示
clouds.png	16.6	512×256	云层	play.png	23	128×128	播放按钮
config.jpg	24	64×128	设置按钮	Process.png	2.78	128×16	加载条
dam.jpg	26.9	512×256	大坝纹理	rador_bg.png	37	128×128	仪表盘背景
exit.jpg	23	64×128	退出按钮	rador_plane.png	7.70	128×128	指示飞机方向的图标
fangwufront.jpg	25.9	128×128	平房的墙	rank.jpg	23	64×128	排行按钮
firebutton.png	14.8	128×128	子弹发射按钮	rank_bg.jpg	245	512×256	排行榜界面背景
Feijione.jpg	124	512×512	第一个飞机的纹理图	rank_number.png	6.6	128×16	数字
Feijitwo.jpg	94	512×512	第二个飞机的纹理图	roofwenli.jpg	41.6	256×256	军火库屋顶纹理
feijithree.jpg	198	512×512	第三个飞机的纹理图	shanding.jpg	52.9	256×256	石头纹理
front_cover.png	7.90	128×64	子弹菜单上的方框	sky.jpg	72.5	1024×512	天空穹纹理
front_frame.png	47.5	512×256	各个界面上的外框	skynight.jpg	51.4	1024×512	夜晚的天空纹理
frontwing2.jpg	21.0	128×128	飞机纹理	smallpingfang.jpg	24.7	128×128	军火库墙纹理
frontwing.jpg	124	128×128	飞机机翼纹理	smallpingfangwuding.jpg	25	128×128	平房屋顶纹理
help.jpg	25.5	64×128	帮助按钮	Sounds_off.png	38.6	256×64	开启特效声音按钮
helpview.png	414	512×1024	帮助界面文字	Sounds_on.png	38.4	256×64	关闭特效声音
icon.png	12.4	64×64	程序图标	start.jpg	25.5	64×128	开始按钮
land.png	281	512×256	菜单滚动背景图片	stop.png	10.5	128×128	停止按钮
light.png	54.7	256×256	灯塔纹理	taiziwenli.jpg	43	512×128	飞机选择场景中的展台纹理
Loading.jpg	127	512×256	加载界面背景	tanke.jpg	44	256×256	坦克纹理
locktexid.png	5	64×64	锁定框	tree2.png	34	128×128	树

续表

图片名	大小（KB）	像素（w×h）	用途	图片名	大小（KB）	像素（w×h）	用途
locktexidaim.png	5	64×64	瞄准器	tree.png	35.4	128×128	树
map_selected_bg.jpg	200	512×256	战役模式下的选关界面	Vibrate_off.png	38.5	256×64	开启特效振动
map_selected_bg_action.jpg	206	512×256	特殊行动模式下的选关界面	Vibrate_on.png	38.3	256×64	关闭特效振动
markarsenal.png	5	32×32	仪表盘上标记军火库位置	war_wzgl.png	110	512×256	王者归来文字说明
Markask.png	5	32×32	仪表盘上标记敌机位置	war_yyxd.png	91	512×256	夜鹰行动文字说明
markplane.png	5	32×32	仪表盘上标记自己位置	war_zjfc.png	70	512×256	终极复仇文字说明
marktanke.png	5	32×32	仪表盘上标记地方坦克和高射炮位置	water.png	11.6	64×64	水纹理
menu_door.jpg	213	512×256	进入设置等界面时，打开的门	xiacengtuceng.jpg	18.5	128×64	下层纹理
menu_two_action_btn_down.png	15	64×64	当前模式为特别行动时显示文字图片	xuebeijing.png	33.5	512×64	血背景
menu_two_action_btn_up.png	12	64×64	当前模式为战役模式时显示文字图片	yuanhuanwenli.png	11.1	512×64	军火库环绕圆环
menu_two_left_down.png	11	64×64	向左按钮按下时	zhonjiantuceng.jpg	34	256×256	土层纹理

（2）接下来介绍的是该游戏中加载地形时使用的是灰度图，将所有灰度图资源放在项目目录中的res/drawable-nodpi文件夹下，其详细情况如表22-2所列。

表22-2　加载山的灰度图片清单

图片名	大小（KB）	像素（w×h）	用途
landform.png	2.83	8×8	加载成地形用于山的左边
landform2.png	2.86	8×8	加载成地形用于接两边的山
landform3.png	2.85	8×8	加载成地形用于山的右边
landform1.png	2.76	8×8	加载成地形用于水面之上

（3）接下来介绍的是该游戏用到的声音和视频资源，将声音资源和视频资源放在项目目录中的res/raw文件夹下，其详细情况如表22-3所列。

表22-3　声音和视频清单

声音/视频文件名	大小（KB）	格　式	用　途
awp_fire.ogg	18	ogg	坦克和高射炮被击中爆炸声音
bullet.mp3	6	mp3	发射子弹声音
explode.ogg	32	ogg	飞机撞山或者死亡的声音
gamebg_music.mp3	7	mp3	背景音乐
ground.ogg	8	ogg	金属声音
m16_fire.ogg	10	ogg	坦克等被击中声音
menubg_music.mp3	470	mp3	菜单时播放的背景音乐
missile.mp3	13	mp3	飞机发射炮弹声音
r700_fire.ogg	15	ogg	大爆炸声音
rotation.MP3	7	mp3	菜单界面下改变选项时的声音
w1200_fire.ogg	10	ogg	坦克发射炮弹声音
logo.3gp	266	3gp	logo 视频

（4）最后介绍的是该游戏用到的构造坦克和飞机模型的obj文件，将这些文件放在项目目录中的assets文件夹下。其详细情况如表22-4所列。

表22-4　模型文件清单

文 件 名	大小（KB）	格　式	用　途
tank_body.obj	4	obj	坦克身体
tank_berral1.obj	3	obj	坦克炮管
feiji11.obj	26	obj	飞机一
feiji22.obj	56	obj	飞机二
feiji33.obj	27	obj	飞机三

22.3　游戏的架构

在介绍该游戏代码的开发之前，首先需要对该游戏的框架进行简单的介绍，使读者更好地理解游戏的开发过程和后面的章节中将要介绍到的内容。

22.3.1　各个类的简要介绍

为了使读者更好地理解各个类的作用，下面将该游戏中的所有类分成9个部分进行介绍，而对于各个类的详细代码将在后面的章节中相继给出。

1．公共类

Activity的实现类Aircraft_Activity类，该类是整个游戏的控制器，也是整个游戏的程序入口， 在该类中需要对游戏中用到的声音等资源进行一些初始化，使用消息处理器进行界面之间的切换。

2．游戏界面相关类

- 游戏界面类GLGameView。

该类负责整个游戏的绘制工作，负责根据游戏情况绘制对应的场景，包括主菜单界面、设置界面、关于界面、帮助界面和排行榜界面等。

- 主菜单绘制类MissileMenuForDraw。

该类为主菜单中物体的具体绘制类，包括导弹上的左右半球，中间圆柱和各个矩形按钮。

- 自定义视频播放控件类MyVideoView。

MyVideoView类是继承自VideoView类，用于启动游戏时的欢迎界面视频播放。

- 游戏线程类KeyThread。

该类为整个游戏逻辑的计算类，包括发射子弹、碰撞检测、虚拟按钮监听、传感器监听、飞机的飞行、飞机发射子弹和炮弹等的模拟，均由该线程类实现。

- 子弹类BulletForControl。

该类为飞机和敌机发射的子弹类，计算其子弹穿行，子弹的碰撞和射击处理等。

- 炮弹类BombForControl。

该类为游戏中的炮弹类，主要是对坦克、高射炮和飞机发射炮弹的模拟和绘制。

- 常量类Constant。

该类为整个程序提供对应的常量、相关静态成员和相关静态方法等。

3．坦克相关类

- 模型的加载类MoXingJiaZai。

该类将由3Dmax制作的模型导出的obj文件加载成对应的顶点数据缓冲，最后绘制出来。

- 模型类Model。

该类为模型类，由模型加载类将模型数据加载进来生成对应的顶点和纹理数据，然后该类根据模型加载类提供的数据生成对应的数据缓冲，最后绘制物体。

- 坦克类TanKe。

该类为坦克类，包括绘制、计算坦克炮管朝向和坦克发射炮弹等。

4．高射炮相关类

- 高射炮的控制类ArchieForControl。

该类为高射炮的控制类，与坦克类一样，包括绘制、计算高射炮朝向和高射炮发射炮弹等。

- 高射炮的绘制类ArchieForDraw。

该类为高射炮的绘制类，将各个小部件组装在一起绘制出整个高射炮。其小部件有炮台和炮管。

- 高射炮的炮台绘制类BarbetteForDraw。

该类将高射炮的炮台绘制出来，炮台由下边圆柱和上圆面组成。

- 高射炮的炮管绘制类BarrelForDraw。

该类将高射炮的炮管绘制出来，其部件有长炮筒、短炮筒、长炮筒圆面和短炮筒圆面等。

5．房屋相关类

- 军火库类Arsenal_House。

该类为军火库类，对军火库模型和军火库的相关生命值背景进行绘制，还提供玩家的飞行器对军火库的锁定计算功能。

- 军火库模型类House。

该类绘制整个军火库，包括房子和围绕房子旋转的颜色圆环等。

- 平房类PlaneHouse。

游戏中没有任何军事作用的平房类，该类绘制两个平房，还有一个圆柱作为其烟囱，平房用于充实游戏界面，使其不单调。

- 三角形类Triangle。

该类用来绘制一个三角形。

- 颜色圆环类Annulus。

该类为绘制围绕军火库旋转的颜色圆环类。

6. 飞机相关类

- 飞机模型类Plane。

该类为绘制飞机的类，飞机由机身、机头、机舱、前机翼、后机翼、上尾翼和螺旋桨等组成。

- 螺旋桨类Airscrew。

该类为螺旋桨的绘制类，螺旋桨由6个梯形组成。

- 圆柱类Column。

该类为圆柱类，用于绘制支持飞机机翼的圆柱。

- 椭球类DrawSpheroid。

用于绘制飞机的机舱等，通过传入相关的尺寸，构造椭球模型，最后绘制。

- 敌机控制类EnemyPlane。

该类为敌机的控制类，控制敌机发射子弹和飞行路线等。

- 纹理矩形Plane_BackWing类、Plane_TopWing类和Plane_Wing类。

Plane_BackWing类用于绘制后机翼，只有顶点和纹理；Plane_TopWing类用于绘制上面的机翼；Plane_Wing类用于绘制机翼。

7. 物体绘制相关类

- 球的绘制类BallTextureByVertex。

该类主要是构造一个球，用于该类的炮弹模型绘制。

- 圆面绘制类CircleForDraw。

该类构造一个圆面，主要用于绘制展台的透明圆面和半透明圆面，以及高射炮的底盘圆面。

- 盒子类CubeForDraw和圆柱类CylinderForDraw。

CubeForDraw类通过绘制前、后、左、右和上5个面来构造一个平房。CylinderForDraw类绘制一个圆柱。

- 大坝类DamForDraw。

该类为绘制场景中的大坝，根据常量类中给出的构造大坝的几个点来构造一个大坝。

- 爆炸矩形类DrawBomb。

该类为绘制爆炸效果类，通过连续放大绘制该纹理矩形，来达到爆炸的效果。当然每次绘制时，都首先计算其朝向，使其朝向摄像机。

- 山地绘制类LandForm。

该类为根据传过来的高度数组构造山地，而山地有倒影，其原理为直接构造对称的顶点，然后在中间绘制半透明水面，既有倒影的效果。

- 圆锥类Light_Tower。

该类主要用来绘制灯塔上的灯和灯塔。

- 物体模型类LoadedObjectVertexNormalTexture。

该类带有顶点坐标、纹理坐标和顶点法向量。

- 数字类NumberForDraw。

该类通过构造10个矩形，每个矩形给予不同的纹理坐标，纹理为一张0～9的数字图片，10个矩形分别对应10个数字的纹理坐标。

- 天空穹类SkyBall。

该类为天空穹的绘制类，为水上部分和水下部分倒影，其原理同样是构造对称的顶点来实现的倒影原理。白天给白天的纹理图片，晚上给夜晚的纹理图片。

- 星空类SkyNight。

该类为星空绘制类，在绘制了外层黑夜星空后，用该类绘制白色圆点的星星。

- 游戏中树的构造类Tree。

该类为传入树形状的纹理，每次绘制时，首先计算其朝向，并对其进行排序，离摄像机近的需要最后绘制，这样使用标准版技术就在3D世界中构造了一棵树。

- 纹理矩形类TextureRect。

该类为纹理矩形的绘制类，包括绘制树和水面等。

8. 相关工具类

- 模型加载类LoadUtil。

该类从obj文件中将模型的相关信息加载进来，然后生成顶点数据、纹理坐标数据和顶点法向量数据，最后构造对应的模型对象。

- 存储系统矩阵状态类MatrixState。

该类实现系统的矩阵变换、矩阵的存储和摄像机位置等，每次绘制物体时，需要从该类得到对应的变换矩阵，是整个3D世界绘制的关键类。

- 法向量记录类Normal。

该类记录顶点的法向量，最后计算顶点平均法向量时会用到。

- 姿态传感器数值计算类RotateUtil。

该类为姿态传感器的静态工具类，提供静态方法进行飞机姿态的计算。

- 着色器的管理类ShaderManager。

该着色器管理类主要是用于加载着色器程序字符串和编译着色器程序，将游戏中用到的所有着色器程序统一管理起来。

- 加载着色器程序的工具类ShaderUtil。

该类加载对应的着色器程序，包括顶点着色器和片元着色器，然后生成对应的程序，供显卡使用。

- 数据库相关类SQLiteUtil。

该类为该游戏的分数记录提供数据存储功能。

22.3.2　游戏框架简介

本小节将对该游戏的整体架构进行介绍，使读者对本游戏有更深的理解，其框架如图22-21所示。



图22-21　游戏框架图

接下来按照程序运行的顺序介绍各个类的作用，其详细步骤如下。

（1）启动游戏，首先会用到Aircraft_Activity类，在Aircraft_Activity中，首先切换到视频播放界面，播放欢迎界面视频。

（2）在视频播放完毕后，创建GLGameView对象，进入加载界面，当所有资源加载完毕，进入的是主菜单界面。

（3）在主菜单界面下可以单击开始按钮进入飞机和场景选择界面，飞机和游戏场景选择界面下，玩家可选择不同的飞行器和不同的游戏模式。单击确定按钮可进入选关界面，在选关界面下选择对应的地图即可进入场景预览界面，场景预览界面下单击停止预览按钮即可进入游戏界面。

（4）在主菜单界面下单击设置按钮，即可进入设置界面，同样单击关于、帮助和排行等按钮可 分别进入相关界面，单击退出按钮可直接退出。所有界面中，除了视频播放界面，均为3D界面，并且均由GLGameView类完成其绘制。

（5）在游戏界面下，玩家改变手机姿态来改变飞机的飞行方向，使用左下角的虚拟按钮来控制其俯仰，使用右下方的发射按钮发射子弹或者导弹，使用左上方的炮弹切换按钮使当前发射的炮弹类型在子弹和导弹之间切换。如果玩家获胜或者被击中，弹出对应的对话框，玩家可根据需求自行选择。

22.4 公共类Aircraft_Activity

从本节开始将正式进入游戏的开发过程，首先介绍的是游戏的控制器Aircraft_Activity类，该类的主要作用是在适当的时间初始化相应的界面，并根据其他界面发送回来的消息切换到用户所需的界面。其开发的详细步骤如下。

（1）首先开发的是Aircraft_Activity的框架，其框架的详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/menu目录下的Aircraft_Activity.java。

```
package com.bn.menu;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘的源代码
@SuppressWarnings("deprecation")
public class Aircraft_Activity extends Activity{
    GLGameView gameView;                                  //主游戏场景
    Handler handler;                                      //消息接收器
    SoundPool soundPool;                                  //声音池
    Vibrator mVibrator;                                   //振动器
    public MediaPlayer bgMusic[]=new MediaPlayer[2];      //游戏背景音乐播放器
        HashMap<Integer,Integer> soundMap;             //存放声音池中的声音 ID 的 Map
        SensorManager mySensorManager;                 //传感器的引用
        private boolean isNoBack;      //返回键屏蔽主要是在欢迎界面播放过程中,屏蔽返回键
        private int flag;                              //判断当前屏幕是否能够旋转的标志位
        public float[] directionDotXY;  //用于记录传感器的数据，directionDotXY[0]表示左右旋转
        public float lr_domain=4;                                  //传感器左右旋转的阈值
        private SensorListener mySensorListener = new SensorListener(){//传感器
        ……//此处省略了初始化传感器的部分代码，将在后面的步骤中给出
        };
    @Override
    public void onCreate(Bundle savedInstanceState){
    ……//此处省略了 onCreate 方法中的部分代码，读者可自行查看随书光盘的源代码
    }
    public void initHandler(){                                     //初始化消息接收器方法
    ……//此处省略了消息接收器初始化的部分代码，读者可自行查看随书光盘的源代码
    }
    public void initScreen(){                                      //初始化屏幕分辨率
    ……//此处省略了初始化屏幕的部分代码，读者可自行查看随书光盘的源代码
    }
    public void initDatebase(){                                    //创建数据库
     String sql="create table if not exists plane(map char(2),grade char(4),time
```

```
    char(4),date char(10));";
31          SQLiteUtil.createTable(sql);                     //建表
32      }
33    public void goTo_StartVideo(){                        //播放视频方法
34    ……//此处省略了播放视频的部分代码，将在后面的步骤中给出
35    }
36    public int getGLVersion(){                            //获取 OpenGLES 所支持的最高版本
37    ……//此处省略了获取 OpenGL ES 支持最高版本的部分代码，读者可自行查看随书光盘的源代码
38    }
39    public void collisionShake() {                        //手机振动的初始化
40            mVibrator=(Vibrator)getApplication().getSystemService(Service.VIBRATOR_
              SERVICE);
41    }
42    public void shake(){                                  //手机振动方法
43        if(0==isVibrateOn){                               //如果开启振动
44            mVibrator.vibrate( new long[]{0,30},-1);      //振动
45        } }
46    public void initSound(){                              //加载声音资源
47    ……//此处省略了加载声音资源的部分代码，读者可自行查看随书光盘的源代码
48    }
49    public void playSound(int sound,int loop){            //播放声音的方法
50    ……//此处省略了播放声音的部分代码，读者可自行查看随书光盘的源代码
51    }
52  @Override
53    public Dialog onCreateDialog(int id){
54    ……//此处省略了对话框相关部分代码，读者可自行查看随书光盘的源代码
55    }
56    @Override
57    protected void onResume(){                            //重写 onResume 方法
58        super.onResume();
59        mySensorManager.registerListener(                 //为传感器注册监听
60            mySensorListener,SensorManager.SENSOR_ORIENTATION,
61            SensorManager.SENSOR_DELAY_UI
62        );}
63    @Override
64    protected void onPause(){                             //重写 onPause 方法
65        super.onPause();
66        mySensorManager.unregisterListener(mySensorListener);//取消注册监听器
67    }
68    public void exitRelease(){                            //退出时需要执行的方法
69        if(flag==0){                                      //关掉旋转屏
70            Settings.System.putInt(this.getContentResolver(),
71                        Settings.System.ACCELEROMETER_ROTATION,0);
72        }
73        System.exit(0);                                   //退出
74    }
75    @Override
76    public boolean onKeyDown(int keyCode, KeyEvent e) {  //键盘监听方法
77    ……//此处省略了键盘监听的部分代码，读者可自行查看随书光盘的源代码
78    }}
```

- 第16-18行为初始化传感器，在该Activity启动后，调用onResume方法，该方法注册传感器监听后，传感器开始工作。同样在该Activity退出后会调用onPause方法，在该方法中传感器取消注册监听后，传感器将会不再占用系统资源。

• 第19-22行为重写的onCreate方法，该方法对相应系统资源进行初始化，然后跳转到视频播放界面，当视频播放完毕后直接进入3D界面的加载界面。

• 第23-25行为消息处理器方法，主要是对在视频播放完毕后发送回来的切换到加载界面消息进行处理。当接收到此消息时，首先创建3D界面，然后切换到加载界面。

• 第26-28行为初始化屏幕分辨率方法，前面章节的案例中已经使用过类似的方法，本章就不再对其进行过多的介绍，请读者自行查阅前一章中对应部分内容，或者查阅随书光盘中的源代码。

• 第29-32行为创建数据库的方法，首先创建数据库，然后创建表。

• 第33-35行为切换到视频播放界面方法，首先将界面切换到视频播放界面，然后对其视频播放完毕添加监听，当视频播放完毕时，首先判断其支持的OpenGL ES的最高版本和当前Android系统版本是否符合要求。如果都符合，则发送切换到加载界面的消息切换到加载界面。

• 第35-51行分别为获取OpenGL ES所支持的最高版本方法、初始化振动器方法、使手机振动方法、初始化声音方法和播放声音方法。

• 第52-55行为初始化对话框方法，该方法为在支持的OpenGL ES的版本不够或者Android系统版本不够时生成对应的对话框。

• 第56-67行分别为重写的onResume方法和onPause方法，在onResume方法中对传感器注册监听，在onPause方法中对传感器取消注册监听，释放资源。

（2）接下来开发的是Aircraft_Activity类中的播放视频的goTo_StartVideo方法，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/menu目录下的Aircraft_Activity.java。

```
public void goTo_StartVideo(){                                   //切换到播放视频界面
    isNoBack=true;                                               //返回键不可用
    setContentView(R.layout.start_video);                        //切换到此 xml 布局
    final MyVideoView myVideoView=(MyVideoView)findViewById(R.id.start_video_videoview);
    myVideoView.setVideoURI(Uri.parse("android.resource://com.bn.menu/" + R.raw.logo));
    myVideoView.start();                                         //开始播放视频
    myVideoView.setOnCompletionListener(new OnCompletionListener(){//添加监听器
            @Override
            public void onCompletion(MediaPlayer mp){            //视频播放完毕时调用方法
                    if(getGLVersion()<2){                        //这里进行 OpenGL ES 测试
                        showDialog(0);                           //弹出对话框,说明不支持该游戏
                    }else if(Build.VERSION.SDK_INT<Build.VERSION_CODES.FROYO){
                        showDialog(1);                           //查看 Android 系统版本是否符合
                    }else
                    handler.sendEmptyMessage(1);                 //如果都符合则进入加载界面
                }}); }
```

说明

该方法为视频播放方法，首先切换到对应界面，然后获得对应控件的引用，将资源文件的uri设置到该控件，最后调用start方法开始播放。

22.5　游戏相关类

前面已经对Aircraft_Activity类的开发进行了介绍，从本节开始将进入游戏开发最主要的环节即游戏界面开发环节。本节先对整个游戏主界面GLGameView类进行介绍，然后再逐步介绍各个与游戏界面相关的类，完成游戏界面的开发。

22.5.1　游戏界面绘制类GLGameView

GLGameView类是整个游戏的绘制类，为整个游戏的呈现类。整个游戏除视频播放界面外其余所有界面均为此类实现，包括菜单界面、关于界面、帮助界面、设置界面、排行榜界面和游戏界面。开发该类的详细步骤如下。

（1）首先开发的是GLGameView类的框架，具体代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的GLGameView.java。

```
package com.bn.gameView;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘中的源代码
public class GLGameView extends GLSurfaceView {
    ……//此处省略了相关成员变量声明的代码，读者可自行查阅光盘的源代码
    public GLGameView(Context context){                    //构造器
    ……//此处省略了初始化传感器的部分代码，读者可自行查阅光盘的源代码
    }
    int shootId=2;                                         //发射按钮的 ID
    int upId=2;                                            //触摸点的 id
        long time;                                         //记录时间
        @Override
        public boolean onTouchEvent(MotionEvent e) {       //触摸事件回调方法
        ……//此处省略了屏幕触摸方法中的部分代码，将在后面的步骤中给出
       }
        private class SceneRenderer implements GLSurfaceView.Renderer  {
          private boolean isFirstFrame=true;               //是否是第一帧
          int plane_hit_id=0;
          public void onDrawFrame(GL10 gl) {
          ……//此处省略了主绘制方法中的部分代码，读者可自行查阅光盘的源代码
          }
          public void onSurfaceChanged(GL10 gl, int width, int height){
             GLES20.glViewport(0, 0, width, height);       //设置视窗大小及位置
            ratio = (float) width / height;                //计算 GLSurfaceView 的宽高比
            ConfigVirtualButtonArea();                     //对按钮的范围进行相应的配置
          }
          public void onSurfaceCreated(GL10 gl, EGLConfig config) {
            ……//此处省略了部分代码，读者可自行查阅光盘的源代码
          }
          public void drawOrthLoadingView(){               //正交投影绘制加载界面
            ……//此处省略了绘制加载界面中的部分代码，将在后面的步骤中给出
          }
          public void drawPerspective(){                   //透视投影绘制游戏场景
```

```
        ……//此处省略了绘制游戏场景方法中的部分代码，将在后面的步骤中给出
    }
    public void drawAll(){                                    //计算当前九宫格
        ……//此处省略了计算九宫格和绘制各物体方法中的部分代码，将在后面的步骤中给出
    }
    public void drawSky(){                                    //绘制天空
        ……//此处省略了绘制天空方法中的部分代码，读者可自行查阅光盘的源代码
    }
    public void drawArchieBombs(){                            //绘制高射炮炮弹
        ……//此处省略了绘制高射炮方法中的部分代码，读者可自行查阅光盘的源代码
    }
    public void drawBullets(){                                //绘制子弹的方法
        ……//此处省略了绘制子弹方法中的部分代码，读者可自行查阅光盘的源代码
    }
    public void drawBombs(){                                  //绘制炮弹的方法
        ……//此处省略了绘制炮弹方法中的部分代码，将在后面的步骤中给出
    }
    public void drawTankBombs(){                              //绘制坦克炮弹
        ……//此处省略了绘制坦克炮弹方法中的部分代码，读者可自行查阅光盘的源代码
    }
    public void drawWater(){                                  //绘制水面的方法
        ……//此处省略了绘制水面方法中的部分代码，读者可自行查阅光盘的源代码
    }
    public void drawLandForm(int rowi,int colj,int rowT,int colT){ //绘制山地
        ……//此处省略了绘制地形方法中的部分代码，将在后面的步骤中给出
    }
    public void draw_number_LandForm(int number) {     //根据编号绘制对应的块
        ……//此处省略了绘制地块方法中的部分代码，将在后面的步骤中给出
    }
    public void drawPlane(float curr_x,float curr_y,float curr_z, //绘制飞机的方法
        float rotationAngle_Plane_X,float rotationAngle_Plane_Y,float
        rotationAngle_Plane_Z){
        ……//此处省略了绘制飞机方法中的部分代码，读者可自行查阅光盘的源代码
    }
    public void drawLightTower(){                             //绘制灯塔
        ……//此处省略了绘制灯塔方法中的部分代码，读者可自行查阅光盘的源代码
    }
    public void drawTanke(int rowi,int colj,int rowT,int colT) {   //绘制坦克
        ……//此处省略了绘制坦克方法中的部分代码，读者可自行查阅光盘的源代码
    }
    public void drawarchie(int rowi,int colj,int rowT,int colT){ //绘制高射炮
        ……//此处省略了绘制高射炮方法中的部分代码，读者可自行查阅光盘的源代码
    }
    public void drawHouse(int ii,int jj,int rowR,int colR) {   //绘制军火库
        ……//此处省略了绘制军火库方法中的部分代码，读者可自行查阅光盘的源代码
    }
    public void drawHousePlane(int ii,int jj,int rowR,int colR){ //绘制普通房子
        ……//此处省略了绘制普通房屋方法中的部分代码，读者可自行查阅光盘的源代码
    }
```

```
      public void drawTree(int ii,int jj,int rowR,int colR) {     //绘制树
        ……//此处省略了绘制树方法中的部分代码，读者可自行查阅光盘的源代码
      }
      public void drawBomb() {                                     //绘制爆炸效果
        ……//此处省略了绘制爆炸效果方法中的部分代码，读者可自行查阅光盘的源代码
      }
      public void drawdam(){                                       //绘制大坝
        ……//此处省略了绘制大坝方法中的部分代码，读者可自行查阅光盘的源代码
      }
      public void drawBaoZhaXiaoguo() {                            //绘制最后飞机爆炸效果
        ……//此处省略了绘制飞机爆炸效果方法中的部分代码，将在后面的步骤中给出
      }
       public void onDrawShiping(){                                //绘制视频界面上的各个按钮
        ……//此处省略了绘制视频界面方法中的部分代码，读者可自行查阅光盘的源代码
        }
      public void onDrawMenu()    {                                //绘制飞机坠毁后的菜单界面
        ……//此处省略了绘制飞机坠毁后菜单界面的部分代码，读者可自行查阅光盘的源代码
      }
      public void onDrawHit(){                                     //绘制飞机被大炮击中时的标识
        ……//此处省略了绘制飞机被击中表示方法中的部分代码，读者可自行查阅光盘的源代码
      }
      public void drawEnemyPlane(){                                //绘制敌机
        ……//此处省略了绘制敌机方法中的部分代码，读者可自行查阅光盘的源代码
      }
      public void drawVirtualIcon(){                               //用正交投影绘制虚拟图标
        ……//此处省略了绘制虚拟按钮方法中的部分代码，读者可自行查阅光盘的源代码
      }
      public void drawMardPlace(){                                 //绘制各个物体的标志位置
        ……//此处省略了绘制物体标志位置方法中的部分代码，读者可自行查阅光盘的源代码
      }
      public void drawMissileMenu() { //绘制导弹菜单选择界面
        ……//此处省略了绘制导弹菜单界面方法中的部分代码，读者可自行查阅光盘的源代码
      }
      public void drawPlaneModel(){
        ……//绘制选择飞机场景的方法
      }
    public void drawCircleStation(float yOffset) {
      ……//绘制选飞机场景中的展台
    }}
  public void loadResource(){                                      //加载所有的资源
  ……//此处省略了加载所有资源方法中的部分代码，将在后面的步骤中给出
  }
  public void init_Shader(){                                       //初始化 shader 方法
      ShaderManager.loadCodeFromFile(getResources()); //初始化字符串
      ShaderManager.compileShader();                               //加载 shader 程序
  }
  public void init_All_Texture(int index){                         //加载纹理的方法
  ……//此处省略了加载纹理方法中的部分代码，读者可自行查阅光盘的源代码
  }
```

```
130        public void init_All_Object(int index) {          //创建所有的对象
131        ……//此处省略了创建对象方法中的部分代码，读者可自行查阅光盘中的源代码
132        }
133        public void initMap(){                            //初始化对应关卡物体
134        ……//此处省略了初始化对应关卡物体方法中的部分代码，将在后面的步骤中给出
135        }
136        public void ConfigVirtualButtonArea(){            //计算虚拟按钮的范围
137        ……//此处省略了计算虚拟按钮范围方法中的部分代码，读者可自行查阅光盘中的源代码
138        }
139        public boolean  onKeyBackEvent(){                 //调用物理键盘的返回键的方法
140        ……//此处省略了键盘监听方法中的部分代码，读者可自行查阅光盘中的源代码
141        }}
```

- 第4-6行为该类的构造器，该构造器主要是初始化一些相关成员变量，计算当前摄像机位置。
- 第11-13行为屏幕触摸事件的处理方法，该游戏中用到了多点触控，而且整个游戏中所有界面由此类完成，所以其逻辑比较复杂，其详细情况将在后面的步骤中给出。
- 第14-115为主要3D界面渲染类，完成整个游戏的绘制等。第120-122行为加载所有资源的方法，一边绘制加载界面，另一边对整个界面中的物体进行加载。第127-132行分别为加载纹理的方法和加载对象的方法。
- 第133-135行为初始化对应关卡中不同物体的方法，如坦克、军火库、敌机和树等。每一关的地图不一样，各个物件的位置也不一样，当地图切换时需要重新加载。
- 第136-138行根据屏幕分辨率初始化后，根据得到的缩放比来计算各个虚拟按钮的位置坐标。
- 第139-141行为在游戏界面下，按下返回键时调用的方法。

说明

此处用到的ShaderManager类，在本书中的第三章已经进行过介绍，因此在本章中对于该类将不再进行介绍，读者可自行查阅随书光盘的源代码。

（2）接下来开发的是该类中的屏幕监听事件处理方法onTouchEvent方法，此处只介绍一下其大致的逻辑框架，其中的详细代码请读者自行查阅随书光盘中的源代码，具体代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的GLGameView.java。

```
1 @Override
2     public boolean onTouchEvent(MotionEvent e) {
3         if(isGameOn){
4             int actionId=e.getAction()&MotionEvent.ACTION_MASK;  //获取触控事件 ID
5             int id=(e.getAction()&MotionEvent.ACTION_POINTER_ID_MASK) //获取主、辅点 id
6                     >>>MotionEvent.ACTION_POINTER_ID_SHIFT;
7             float x=e.getX(id);                                   //得到对应触控 id 下的坐标
8             float y=e.getY(id);
9             switch(actionId) {
10                case MotionEvent.ACTION_POINTER_DOWN:            //辅控点
11                case MotionEvent.ACTION_DOWN:                    //主控点
12                ……//此处为对游戏中和在游戏时弹出的对话框等进行监听处理
13                break;
14              case MotionEvent. ACTION_POINTER_UP:              //辅控点抬起
15                ……//此处未在游戏界面下屏幕触摸抬起时的处理代码，此处不再对其进行详细介绍
16                break;
17              case MotionEvent.ACTION_UP:                       //主控点抬起
18                ……//此处未在游戏界面下屏幕触摸抬起时的处理代码，此处不再对其进行详细介绍
19                break;
20          }}else{                                              //游戏还没有开始，处于其他界面时
21                float x=e.getX();
22                float y=e.getY();
23                switch(e.getAction()){
24                case MotionEvent.ACTION_DOWN:                    //触摸按下
25                    if(1==isMenuLevel) {                         //主菜单
26                    ……//此处为在主菜单界面下的屏幕监听处理代码，此处不对其进行详细介绍
27                    }
28                    else if(2==isMenuLevel){                     //飞行器和游戏模式选择界面
29                    ……//此处未在飞机和模式选择界面下的监听处理代码
30                    }
31                    else if(3==isMenuLevel) {                    //对应模式下的地图选择界面
32                    ……//此为关卡选择界面下的触摸处理代码，此处不对其进行详细介绍
33                    }}
34                    break;
35                case MotionEvent.ACTION_MOVE:                    //触摸移动事件处理
36                ……//此处省略了在其他界面下的触摸事件的处理，此处不对其进行详细介绍
37                    break;
38                case MotionEvent.ACTION_UP:                      //触摸抬起事件处理
39                ……//此处省略了在其他界面下的触摸抬起事件的处理，此处不对其进行详细介绍
40                    break;
41                }
42                mPreviousY=y;
43          }
44          return true;
45      }
```

- 第3-20行为在游戏界面下，使用多点触控来控制子弹的发射和飞机的俯仰，根据对应模式下对弹出的对话框进行相应的处理，以及

对视频播放界面下的各个按钮的逻辑处理等。

- 第24-34行为其他界面下，主要包括主菜单界面、声音设置界面、排行榜界面、关于界面、帮助界面、飞行器选择界面、关卡选择界面。根据所处界面下和触摸的坐标位置进行对应的逻辑处理。
- 第35-37行为在其他界面下对触摸移动事件的处理。
- 第38-40行为在其他界面下对触摸抬起事件的处理。如在飞行器选择界面下，只有当按下和抬起时均在同一个按钮上时才表示按下了对应的按钮，然后根据按下此按钮时的功能进行逻辑处理。

（3）接下来开发的是渲染器类中的绘制加载界面的drawOrthLoadingView方法。在没有加载完毕时，均只绘制加载界面下的背景矩形和加载进度条，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的GLGameView.java。

```
1 public void drawOrthLoadingView(){
2         if(isFirstFrame) {                                         //如果是第一帧
3              MatrixState.pushMatrix();                             //保护现场
4             MatrixState.setProjectOrtho(-1, 1, -1, 1, 1, 10);      //设置正交投影
5             MatrixState.setCamera(0, 0, 1, 0, 0,-1, 0, 1, 0);      //设置摄像机
6             MatrixState.copyMVMatrix();                            //初始化矩阵
7             loadingView.drawSelf(tex_loadingviewId);               //绘制背景矩形
8             MatrixState.popMatrix();                               //恢复现场
9              isFirstFrame=false;                                   //标记已经不是第一帧了
10            }else{                                                 //如果不是第一帧
11               MatrixState.pushMatrix();
12                 MatrixState.setProjectOrtho(-1, 1, -1, 1, 1, 10);   //设置正交投影
13                 MatrixState.setCamera(0, 0, 1, 0, 0,-1, 0, 1, 0);   //设置摄像机
14                 MatrixState.copyMVMatrix();                         //初始化变换矩阵
15                 MatrixState.pushMatrix();                           //保护现场
16                 MatrixState.translate(-2+2*load_step/(float)40, -1+0.05f, 0f);
17                 processBar.drawSelf(tex_processId);                 //绘制加载条
18                 MatrixState.popMatrix();                            //恢复现场
19                 loadingView.drawSelf(tex_loadingviewId);            //绘制背景图
20                 MatrixState.popMatrix();                            //恢复现场
21                 loadResource();                                     //加载资源
22                 return;
23            }}
```

- 第2-10行为绘制第一帧时，只绘制加载条和加载界面下的背景图片，因为只有在整个onDrawFrame方法执行完毕时，才在屏幕上显示

图像，第一帧不对物体进行加载，减少黑屏时间，从第二帧开始才对场景中的物体进行加载。

- 第10-11行为从第二帧开始，每次需要绘制加载界面，同时对场景中物体进行加载，每一帧加载一部分，直到完全加载完毕才开始绘制游戏场景。

> **说明**
>
> 此处用到的MatrixState类，在本书第4章已经进行过介绍，因此对于该类此处不再介绍，读者可自行查阅随书光盘的源代码。

（4）接下来开发的是渲染器类中的游戏场景绘制方法drawPerspective，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的GLGameView.java。

```
public void drawPerspective(){
        MatrixState.setProjectFrustum(-ratio, ratio, -1, 1, 3, 40000);//设置透视投影
        MatrixState.pushMatrix();                          //保护矩阵
        synchronized(lock) {  //这里进行相关变量的复制,用于不同线程之间的不同步
            curr_cx=cx;                                    //复制摄像机位置
            curr_cy=cy+2;
            curr_cz=cz;
            curr_tx=tx;                                    //复制摄像机目标位置
            curr_ty=ty;
                curr_tz=tz;
                curr_upx=upX;                              //复制 up 向量
                curr_upy=upY;
                curr_upz=upZ;
                curr_PlaneX=PLANE_X;                       //赋值飞机位置
                curr_PlaneZ=PLANE_Z;
```

```
16              curr_PlaneY=PLANE_Y;
17              curr_rot_Plane_X=rotationAngle_Plane_X;      //赋值飞机的方向
18              curr_rot_Plane_Y=rotationAngle_Plane_Y;
19              curr_rot_Plane_Z=rotationAngle_Plane_Z;
20              Matrix.setRotateM(fa, 0, curr_rot_Plane_Y, 0, 1, 0);//得到旋转矩阵
21        YB[0]=(float) (Math.sin(Math.toRadians(-curr_rot_Plane_Z)));
                                                          //得到绕 z 轴旋转的 x 分向量
22        YB[1]=(float) (Math.cos(Math.toRadians(-curr_rot_Plane_Z))*
                                                          //得到绕 z 轴旋转的 y 分向量
23                     Math.cos(Math.toRadians(rotationAngle_Plane_X)));
24        YB[2]=(float) (Math.cos(Math.toRadians(-curr_rot_Plane_Z))*
                                                          //得到绕 z 轴旋转的 z 分向量
25                     Math.sin(Math.toRadians(rotationAngle_Plane_X)));
26              YB[3]=1;
27              Matrix.multiplyMV(resultUp, 0, fa, 0, YB, 0);    //得到up向量
28              MatrixState.setCamera(curr_cx,curr_cy,curr_cz,curr_tx,curr_ty,curr_tz,
29                    resultUp[0],resultUp[1],resultUp[2]);   //设置摄像机的位置
30              MatrixState.copyMVMatrix();                        //初始化矩阵
31              MatrixState.translate(-20*(float)Math.sin(Math.toRadians(curr_rot_
                Plane_Z/4)),
32                    -20*(float)Math.cos(Math.toRadians(curr_rot_Plane_Z/4)),0);
33              copybaozhaList.clear();                      //复杂爆炸效果列表
34              for(DrawBomb db:baoZhaList){
35                  copybaozhaList.add(db);
36              }
37              cop_archie_bomb_List.clear();                //复制高射炮子弹
38              for(BombForControl db:archie_bomb_List){
39                  cop_archie_bomb_List.add(db);
40              }
41              cop_bomb_List.clear();                       //复制炮弹列表
42              for(BombForControl db:bomb_List){
43                  cop_bomb_List.add(db);
44              }
45              cop_archie_List.clear();                     //高射炮炮弹复制
46              for(ArchieForControl db:archie_List) {
47                  cop_archie_List.add(db);
48              }
49              cop_bullet_List.clear();                     //子弹复制
50              for(BulletForControl db:bullet_List) {
51                  cop_bullet_List.add(db);
52              }
53              copy_tank_bomb_List.clear();                 //炮弹复制
54              for(BombForControl db:tank_bomb_List){
55                  copy_tank_bomb_List.add(db);
56              }}
57           lightAngle+=2;                                  //灯塔的旋转角度
58         drawAll();       //计算当前绘制格子范围,并调用相应的方法绘制对应物体
59         MatrixState.popMatrix();                          //恢复矩阵
60      }
```

- 第5-27行为复制摄像机相关的变量，然后计算up向量，up向量是绕y轴旋转后再绕x轴旋转而得，其计算过程请见注释。
- 第28-59行为设置摄像机位置，并初始化矩阵，然后复制游戏中装有各个物体的列表，最后调用drawAll方法计算其绘制的地块范围并绘制所有物体。

（5）接下来开发的是绘制所有物体的drawAll方法，具体代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的GLGameView.java。

```
public void drawAll(){
        int rowi=0;                                      //起始行列
        int colj=0;
        int colT=CELL_SIZE;                              //终止行列
        int rowT=CELL_SIZE;
        int col=(int)(curr_PlaneX/WIDTH_LALNDFORM);//-CELL_SIZE/2;//得到当前所在的行列
        int row=(int)(curr_PlaneZ/HEIGHT_LANDFORM);//-CELL_SIZE/2;
        int rcCount=2;                                   //向前飞时后面要绘制的地块数
        curr_rot_Plane_Y%=360;                           //飞行方向归正到 0~360 内
            if(curr_rot_Plane_Y<0){
                curr_rot_Plane_Y=360+curr_rot_Plane_Y;
            }
            if((curr_rot_Plane_Y>=0&&curr_rot_Plane_Y<=45)|| //如果原 z 轴负方向 45° 角内飞行
                (curr_rot_Plane_Y>=315&&curr_rot_Plane_Y<=360)) {
                if(curr_PlaneZ<0) {                      //在背岛而飞时，全岛都不绘制
                    rowi=0;
                    rowT=1;
                    colj=0;                              //所有地块都得绘制
                    colT=MapArray[mapId].length;
                }else if(curr_PlaneZ>MapArray[mapId].length*WIDTH_LALNDFORM){
                    rowi=MapArray[mapId].length-CELL_SIZE/2;     //所有地块都得绘制
                    rowT=MapArray[mapId].length;
                    colj=0;
                    colT=MapArray[mapId].length;
                }
                else{                                    //否则在岛中间，就只绘制飞机前面地块
                    rowi=row-CELL_SIZE/2;                //起始行为当前行向后推移
                    if(rowi<0){                          //不能移出范围
                        rowi=0;
                    }
                    rowT=row+rcCount;     //终止行为当前行推移到需要绘制的飞机后面的块数
                    if(rowT>MapArray[mapId].length){     //规范到整个地块以内
                     rowT=MapArray[mapId].length;
                    }else if(rowT<0){
                        rowT=0;
                    }
                    colj=col-CELL_SIZE/2;                //列数调整,规范到绘制的块数以内
                    colT=col+CELL_SIZE/2;
                    if(colj<0){
                        colj=0;
                    }else if(colj>MapArray[mapId].length){
                        colj=MapArray[mapId].length;
                    }
                    if(colT<0){
```

```
45                  colT=0;
46              }else if(colT>MapArray[mapId].length){
47                  colT=MapArray[mapId].length;
48              }}}
49          else if(curr_rot_Plane_Y>=45&&curr_rot_Plane_Y<135){
50          ……//此处省略了飞机原 x 轴负方向 45° 角范围飞行时裁剪的处理代码
51          }else if(curr_rot_Plane_Y>=135&&curr_rot_Plane_Y<225){
52          ……//此处省略了飞机原 z 轴正方向 45° 角范围飞行时裁剪的处理代码
53          }else{
54          ……//此处省略了飞机原 x 轴正方向 45° 角范围飞行时裁剪的处理代码
55          }
56            drawSky();                                          //绘制天空
57           drawdam();                                           //绘制大坝
58           drawLandForm(rowi,colj,rowT,colT);                   //绘制岛
59           drawWater();                                         //绘制水
60           drawTanke(rowi,colj,rowT,colT);                      //绘制坦克
61           drawarchie(rowi,colj,rowT,colT);                     //绘制高射炮
62           drawHouse(rowi,colj,rowT,colT);                      //绘制军火库
63            drawHousePlane(rowi,colj,rowT,colT);                //绘制房子
64            drawBombs();                                        //绘制炮弹
65            drawArchieBombs();                                  //绘制高射炮炮弹
66            drawTankBombs();                                    //绘制坦克发射的炮弹
67            drawEnemyPlane();                                   //绘制敌机
68            drawPlane(curr_PlaneX,curr_PlaneY,curr_PlaneZ,      //绘制玩家操控飞机
69                  curr_rot_Plane_X,curr_rot_Plane_Y,curr_rot_Plane_Z);
70           drawLightTower();                                    //绘制灯塔
71            drawTree(rowi,colj,rowT,colT);                      //绘制树
72           drawBomb();                                          //绘制爆炸效果
73            drawBullets();                                      //绘制子弹
74           drawBaoZhaXiaoguo();                                 //军火库爆炸效果
75        }
```

- 第13-55行为根据飞机飞行的方向，来计算此时应该绘制的地块范围，当飞机在背岛而飞、进岛而飞和岛中飞行时，分别绘制的地块范围不一样，在岛中飞行时，只绘制飞机前面的格子，后面的只向后推移一定的格子范围，对前面绘制的格子进一步裁剪，这样能绘制更少的物体而取得同样的效果，增加帧速率。
- 第56-74行为分别调用对应的方法绘制对应物体，在绘制某些需要剪裁绘制的物体时，需传入计算好的行列范围，当对应物体不在此格子范围内时则不绘制，上方的天空穹正好把没有绘制的部分给遮挡了。

说明

该处只介绍了当飞机沿着z轴负方向45°范围内飞行时对地块的剪裁计算方法，在其他方向飞行时的剪裁计算原理是一样的，此处

就不再对其进行详细的介绍了，请读者自行查阅随书光盘中的源代码。

（6）接下来开发的是渲染器类中的根据编号绘制地块的draw_number_LandForm方法，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的GLGameView.java。

```
public void draw_number_LandForm(int number){
switch (number){
    case 0:                                          //编号为 0,、1、2 时直接调用绘制
    case 1:                                          //此几个地块的左上角为坐标原点
    case 2:terrain[number].drawSelf(0,tex_terrain_shandingId,tex_terrain_tuceng_Id,
            tex_terrain_caodiId,tex_terrain_shitouId, LOTHight,height_span_LOT);
    break;
    case 3:                                          //3 号地块只是纹理不一样
        terrain[number].drawSelf(1,tex_terrain_shandingId, tex_terrain_ tuceng_Id,
            tex_terrain_caodiId,tex_terrain_shitouId, HillHight,height_span_Hill);
            break;
        case 4:                                      //4 号地块由 0 号地块旋转 90°得到
            MatrixState.translate(0, 0, WIDTH_LALNDFORM);
            MatrixState.rotate(90, 0, 1, 0);
            terrain[0].drawSelf(0,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                    tex_terrain_caodiId,tex_terrain_shitouId, LOTHight,height_span_
                    LOT);
            break;
        case 5:                                      //5 号地块由 0 号地块旋转 180°得到
            MatrixState.translate(WIDTH_LALNDFORM, 0, WIDTH_LALNDFORM);
            MatrixState.rotate(180, 0, 1, 0);
            terrain[0].drawSelf(0,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                    tex_terrain_caodiId, tex_terrain_shitouId,LOTHight,height_span_
                    LOT);
            break;
        case 6:                                      //6 号地块由 0 号地块旋转 270°得到
            MatrixState.translate(WIDTH_LALNDFORM, 0, 0);
            MatrixState.rotate(270, 0, 1, 0);
            terrain[0].drawSelf(0,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                    tex_terrain_caodiId,tex_terrain_shitouId, LOTHight,height_span_
                    LOT);
            break;
        case 7:                                      //7 号地块由 1 号地块旋转 90°得到
            MatrixState.translate(0, 0, WIDTH_LALNDFORM);
            MatrixState.rotate(90, 0, 1, 0);
            terrain[1].drawSelf(0,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                    tex_terrain_caodiId,tex_terrain_shitouId, LOTHight,height_span_LOT);
            break;
        case 8:                                      //8 号地块由 1 号地块旋转 180°得到
            MatrixState.translate(WIDTH_LALNDFORM, 0, WIDTH_LALNDFORM);
            MatrixState.rotate(180, 0, 1, 0);
            terrain[1].drawSelf(0,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                    tex_terrain_caodiId,tex_terrain_shitouId, LOTHight,height_span_LOT);
            break;
        case 9:                                      //9 号地块由 1 号地块绕 Y 轴旋转 27°得到
            MatrixState.translate(WIDTH_LALNDFORM, 0, 0);
            MatrixState.rotate(270, 0, 1, 0);
            terrain[1].drawSelf(0,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                    tex_terrain_caodiId,tex_terrain_shitouId, LOTHight,height_span_LOT);
            break;
        case 10:                                     //10 号地块由 2 地块旋转 90°得到
            MatrixState.translate(0, 0, WIDTH_LALNDFORM);
```

```java
            MatrixState.rotate(90, 0, 1, 0);
            terrain[2].drawSelf(0,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                    tex_terrain_caodiId,tex_terrain_shitouId, LOTHight,height_span_LOT);
            break;
        case 11:                                    //11 号地块由 2 地块旋转 180°得到
            MatrixState.translate(WIDTH_LALNDFORM, 0, WIDTH_LALNDFORM);
            MatrixState.rotate(180, 0, 1, 0);
            terrain[2].drawSelf(0,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                    tex_terrain_caodiId,tex_terrain_shitouId, LOTHight,height_span_LOT);
            break;
        case 12:                                    //12 号地块由 2 地块旋转 270°得到
            MatrixState.translate(WIDTH_LALNDFORM, 0, 0);
            MatrixState.rotate(270, 0, 1, 0);
            terrain[2].drawSelf(0,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                    tex_terrain_caodiId, tex_terrain_shitouId,LOTHight,height_span_LOT);
            break;
        case 13:                                    //高平面，为一个与地块大小一样的纹理矩形
            MatrixState.translate(WIDTH_LALNDFORM/2, LAND_HIGHEST,
                        WIDTH_LALNDFORM/2);
            MatrixState.rotate(-90, 1, 0, 0);
            terrain_plain.drawSelf(tex_terrain_caodiId);
            break;
        case 15:                                    //水面上的山
            terrain[4].drawSelf(0,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                tex_terrain_caodiId,tex_terrain_shitouId, waterHillHight,height_
                span_Water);
            break;
        case 16:                                    //地块上的山
            terrain[5].drawSelf(1,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                tex_terrain_caodiId, tex_terrain_shitouId, HillHight,height_span_Hill);
            break;
        case 17:                                    //中间山 17 号地块
            terrain[6].drawSelf(1,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                tex_terrain_caodiId, tex_terrain_shitouId, HillHight,height_span_Hill);
            break;
        case 18:                                    //18 号地块由左接山地块旋转 270°所得
            MatrixState.translate(WIDTH_LALNDFORM, 0, 0);
            MatrixState.rotate(270, 0, 1, 0);
            terrain[3].drawSelf(1,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                tex_terrain_caodiId, tex_terrain_shitouId, HillHight,height_span_Hill);
            break;
        case 19:                                    //19 号地块由中间山旋转 270°所得
            MatrixState.translate(WIDTH_LALNDFORM, 0, 0);
            MatrixState.rotate(270, 0, 1, 0);
            terrain[6].drawSelf(1,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                tex_terrain_caodiId, tex_terrain_shitouId, HillHight,height_span_Hill);
            break;
        case 20:                                    //20 号地块由右接地块山旋转 270°所得
            MatrixState.translate(WIDTH_LALNDFORM, 0, 0);
            MatrixState.rotate(270, 0, 1, 0);
            terrain[5].drawSelf(1,tex_terrain_shandingId,tex_terrain_tuceng_Id,
                tex_terrain_caodiId, tex_terrain_shitouId, HillHight,height_span_Hill);
            break;
```

```
        case 21:                                          //绘制跑道
            MatrixState.translate(WIDTH_LALNDFORM/2, LAND_HIGHEST,
                WIDTH_LALNDFORM/2);
            MatrixState.rotate(-90, 1, 0, 0);
            terrain_plain.drawSelf(tex_damId);
            break;
        } }
```

说明

岛的边沿为3块不同形状的基本地形，通过旋转和移动搭建而成。岛上的山是通过加载灰度图生成，有两种山，一种是横向的，另外一种是纵向的。纵向的山为使用横向的山旋转而得。整座山由3块可以对接上的基本地块山对接而成。

（7）接下来开发的是渲染器类中的飞机爆炸效果的绘制方法drawBaoZhaXiaoguo，具体代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的GLGameView.java。

```
public void drawBaoZhaXiaoguo(){
        GLES20.glDisable(GLES20.GL_CULL_FACE);                  //关闭背面剪裁
        if(isnoCrash) {                                         //如果是飞机炸毁
            isFireOn=false;
            fireButton.isButtonDown=0;                          //按钮不再变化
            up_button.isButtonDown=0;                           //按钮不再变化
            down_button.isButtonDown=0;                         //按钮不再变化
            keyState=keyState&0xc;                              //设置按键标志的对应比特值
            upId=2;                                             //对应按下的触控 id 都归位
                shootId=2;
                GLES20.glEnable(GLES20.GL_BLEND);               //开启混合
                GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA,         //设置混合因子
                        GLES20.GL_ONE_MINUS_SRC_ALPHA);
                MatrixState.pushMatrix();                       //保护矩阵
                MatrixState.translate(curr_PlaneX,curr_PlaneY,curr_PlaneZ);
                MatrixState.scale(BaoZha_scal, BaoZha_scal, BaoZha_scal);//设置放大的参数
                MatrixState.rotate(-90, 1, 0, 0);               //旋转
                bombRect.drawSelf(baoZhaXiaoguo);               //绘制爆炸矩形
                MatrixState.popMatrix();                        //恢复矩阵
                GLES20.glDisable(GLES20.GL_BLEND);              //关闭混合
            }
            GLES20.glEnable(GLES20.GL_CULL_FACE);               //打开背面剪裁
        }
```

说明

飞机的爆炸效果是在飞机被击毁时，首先使飞机掉落到地面，然后使摄像机绕着飞机旋转并拉伸，最后添加爆炸效果，整个爆炸效果就是将一个爆炸效果图片不断放大绘制，最后瞬间缩小到无的一个过程。

（8）接下来开发的是该类中的加载所有资源的loadResource方法，其详细代码如下。

代码位置：见随书光盘中源代码/第22章

AircraftAttack/src/com/bn/gameView目录下的GLGameView.java。

```
public void loadResource(){
    switch(load_step){                              //根据当前加载情况
    case 0:                                         //如果是 0
        init_All_Texture(load_step);                //加载纹理
        init_All_Object(load_step);                 //创建对应的对象
        load_step++;                                //加载情况标识自加
        break;
    case 1:                                         //下一步
        init_All_Texture(load_step);                //加载纹理
            init_All_Object(load_step);             //创建对应的对象
            load_step++;
            break;
        ……//以下都采用同样的方式加载，此处省略了部分代码，请自行查阅随书光盘中源代码
        case 9:                                     //最后到 9
            init_All_Texture(load_step);            //加载纹理
            init_All_Object(load_step);             //创建对应的对象
            isLoadedOk=true;                        //加载完毕标志标记为加载完毕
            break;                                  //此时可开始绘制游戏场景了
        }}
```

说明

此方法为加载所有资源的方法，在加载界面下每刷一帧就调用此方法加载一些相关资源，直到最后加载完所有资源。而加载纹理和创建对象的方法也同样，传入load_step，根据此变量加载对应的部分资源。此处就不再对这两个方法进行详细的介绍了，请读者自行查阅随书光盘中源代码。

（9）最后开发的是初始化各物体列表的initMap方法，该方法主要是在对应模式的不同地图下，重新初始化其对应物体列表，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的GLGameView.java。

```
public void initMap(){                                          //每一个地图都不一样
        tankeList.clear();                                      //坦克清零
        archie_List.clear();                                    //高射炮清零
        bomb_List.clear();                                      //炮弹清零
        archie_bomb_List.clear();                               //高射炮炮弹清零
        bullet_List.clear();                                    //飞机发射的子弹清零
        enemy.clear();                                          //敌机数组清零
        arsenal.clear();                                        //军火库清零
        houseplane.clear();                                     //平房清零
            treeList.clear();                                   //树
            for(int i=0;i<enemy_plane_place[mapId].length;i++){       //创建敌机
                enemy.add(new EnemyPlane(this,plane,enemy_plane_place[mapId][i][0],
                        enemy_plane_place[mapId][i][1],enemy_plane_place[mapId][i][2],
                        enemy_plane_place[mapId][i][3],enemy_plane_place[mapId][i][4],
                        enemy_plane_place[mapId][i][5],
                        backgroundRect,   numberRect,mark_placeRect,mark_lock,i
                )); }
            for(int i=0;i<ArchieArray[mapId][1].length/2;i++){        //创建坦克
                tankeList.add(new TanKe(this,bullet_ball,tanke_body,tanke_gun,
```

```
20                  new float[]{ArchieArray[mapId][1][i*2]*WATER_WIDTH,LAND_
                    HIGHEST,
21                  ArchieArray[mapId][1][i*2+1]*WATER_WIDTH},
22          (int)ArchieArray[mapId][1][i*2],(int)ArchieArray[mapId][1][i*2+1],
            backgroundRect,
23                  numberRect,mark_placeRect,mark_lock));
24      }
25      for(int i=0;i<ArchieArray[mapId][0].length/2;i++){   //将高射炮放入列表中
26          archie_List.add(new ArchieForControl (this, barrel, barbette, cube,
            bullet_ball,
27                  new float[]{ArchieArray[mapId][0][i*2]*WIDTH_LALNDFORM,
28                  LAND_HIGHEST+barbette_length/2,
29                  ArchieArray[mapId][0][i*2+1]*WIDTH_LALNDFORM},
30                  (int)ArchieArray[mapId][0][i*2+1],(int)ArchieArray[mapId]
                    [0][i*2],
31                  backgroundRect,   numberRect,mark_placeRect,mark_lock)); }
32      for(int i=0;i<ArchieArray[mapId][2].length/2;i++){   //创建军火库
33      arsenal.add(new Arsenal_House(house, ArchieArray[mapId][2][2*i]*WIDTH_
        LALNDFORM,
34              LAND_HIGHEST, ArchieArray[mapId][2][2*i+1]*WIDTH_LALNDFORM,
35              mark_placeRect,mark_lock,
36              (int)ArchieArray[mapId][2][2*i],(int)ArchieArray[mapId][2][2*i+1]));
                                                               //创建军火库
37      }
38      for(int i=0;i<ArchieArray[mapId][11].length/4;i++){  //创建树
39          treeList.add(new Tree(treeRect,ArchieArray[mapId][11][i*4]*WIDTH_
            LALNDFORM,
40                  LAND_HIGHEST+treeHeight/2-5,
41                  ArchieArray[mapId][11][i*4+1]*WIDTH_LALNDFORM,treeTexId_2,
42                  (int)ArchieArray[mapId][11][i*4],(int)ArchieArray[mapId]
                    [11][i*4+1]));
43          treeList.add(new Tree(treeRect,ArchieArray[mapId][11][i*4+2]*WIDTH_
            LALNDFORM,
44                  LAND_HIGHEST+treeHeight/2-5,
45                  ArchieArray[mapId][11][i*4+3]*WIDTH_LALNDFORM,treeTexId,
46                  (int)ArchieArray[mapId][11][i*4+2],(int)ArchieArray[mapId]
                    [11][i*4+3]
47          )); }
48      for(int i=0;i<ArchieArray[mapId][4].length/2;i++){   //创建房屋
49      houseplane.add(new PlaneHouse(ArchieArray[mapId][4][2*i]*WIDTH_LALNDFORM,
50                  LAND_HIGHEST+house_height/2,
51                  ArchieArray[mapId][4][2*i+1]*WIDTH_LALNDFORM,
52      housePlane,chimney,(int)ArchieArray[mapId][4][2*i],(int)ArchieArray
        [mapId][4][2*i+1])); }
53      dam=null;
54      if(ArchieArray[mapId][5].length>0) {              //创建大坝
55          dam=new DamForDraw(LAND_HIGHEST-20,30,90,150,
56                  ShaderManager.getOnlyTextureShaderProgram());
57      }}
```

- 第2-10行为清空对应列表，然后将对应地图下的物体添加进对应列表，其中每一种物体均有其对应的常量数组，根据该物体的位置数组添加对应物体。
- 第11-17行为创建敌机，需要传入相应飞机模型，敌机初始位置、敌机高度、血的背景矩形、 锁定框等。而其位置信息都是根据常量数组中给出的数据来创建的。
- 第18-31行为根据数组中坦克、高射炮的数据创建坦克列表和高射炮列表，传入其对应的模型、位置数据、血的背景矩形、锁定框

等。

• 第32-37行同样是根据给出的常量数组中数据创建军火库，传入军火库模型位置数据、血的背景矩形、锁定框等。

• 第38-47行为创建树的列表，该游戏中有两种树。在进行绘制时，每次绘制前都要对其进行排序，离摄像机远的树要先进行绘制。

• 第48-52行为创建房屋，需要传入房屋模型、位置数据等。

• 第54-56行为创建大坝，该游戏中只有第一个地图有大坝，在其对应数组中如果没有给予相关数据则说明没有大坝。

22.5.2　游戏线程类KeyThread

该类为游戏的主线程类，主要负责整个游戏的逻辑计算，飞机的飞行模拟，敌方坦克、高射炮、飞机发射子弹的模拟，碰撞检测模拟等，开发该类的详细步骤如下。

（1）首先开发的是该类的代码框架，其详细代码如下。



代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的KeyThread.java。

```
package com.bn.gameView;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘中的源代码
public class KeyThread extends Thread {
    public boolean flag_go;                                       //线程标志位
    private boolean isPlaneNoUp;                                  //飞机不再上升标志位
    GLGameView gv;                                                //主场景类的引用
    private float oldTimeBullet=0;                                //用于记录上次发射子弹的时间
    private float oldTimeBomb=0;                                  //用于记录上次发射炸弹的时间
    int time;
        public float planeY;                                      //飞机坠毁处的地面高度
        int playId=1;                                             //视频播放
        public boolean isno_adjust;                               //是否调整飞机方向
        int playIdArray;                                          //数组下标
        public KeyThread(GLGameView gv){                          //构造器
            this.gv=gv;
            flag_go=true;
            tx=PLANE_X=MapArray[mapId].length*WIDTH_LALNDFORM/2;//飞机位置初始化
            ty=PLANE_Y;
            tz=PLANE_Z=MapArray[mapId].length*WIDTH_LALNDFORM/2;
            isnoVideo=true;
        }
        @Override
        public void run(){                                        //重写的 run 方法
        ……//场景预览和特效行动模式下玩家输赢情况的处理模块
        ……//玩家的飞机坠毁或者成功通关处理模块
        ……//键盘监听处理模块，即飞机方向计算处理模块
        ……//碰撞检测处理模块
```

```
        ……/其他物体模拟模块
        ……//其他界面下的处理模块，此处不对其进行介绍，请读者自行查阅随书光盘中的源代码
        }}
        public boolean isnoHitHill(float fjX,float fjY,float fjZ,float yAngle,float xAngle){
        ……//此处省略了飞机碰撞检测的部分代码，将在后面的步骤中给出
        }
        public  boolean isYachtHeadCollectionsWithLand(float ctx,float ctyh,float ctz){
        ……//此处省略了碰撞检测的具体计算的部分代码，将在后面的步骤中给出
        }
        public static float isYachtHeadCollectionsWithLandPaodan(float ctx,float ctyh,float ctz){
        ……//此处省略了飞行炮弹的和山地的碰撞检测的部分代码，将在后面的步骤中给出
        }
        public static boolean isInTriangle(
            float x1,float y1,                                    //三角形第一个点的 X、Y 坐标
            float x2,float y2,                                    //三角形第二个点的 X、Y 坐标
            float x3,float y3,                                    //三角形第三个点的 X、Y 坐标
            float dx,float dy) {
        ……//此处省略了判断一个点是否在三角形内的方法的部分代码，将在后面的步骤中给出
        }
        public static float fromXZToY  (//计算由 3 个点 0、1、2 确定的平面在指定 X、Z 坐标处的高度
            float tx0,float ty0,float tz0,                        //确定平面的点 0
            float tx1,float ty1,float tz1,                        //确定平面的点 1
            float tx2,float ty2,float tz2,                        //确定平面的点 2
            float ctx,float ctz){                                 //船头的 X、Z 坐标
            ……//此处省略了计算对应点高度的部分代码，将在后面的步骤中给出
        }}
```

- 第22-30行为该线程类中重写的run方法，该游戏中所有的逻辑处理均放在此方法中，对于各个功能模块，其代码将会在后面的内容中给出。
- 第31-33行为计算飞机碰撞检测的情况，根据飞机的高度计算出飞机的机头高度，判读其是否低于所在点的山地高度，若低于山高度则表示发生了碰撞，生命值减少，同时使飞机上升到一个不碰撞到山的高度。
- 第34-36行为根据一点的坐标计算该点垂直方向上山地对应点的高度，以此来判断其是否和山发生了碰撞。
- 第37-39行为计算炮弹是否碰到山地，如果炮弹碰撞到山地，则将其自动爆炸并清除出炮弹列表，此炮弹被销毁。
- 第40-46行为判断一个点是否在一个三角形内的方法。第47-53行为计算一个点的垂直方向上该点的地面高度。

（2）接下来开发的是该类中run方法下场景预览和特效行动模式下玩家输赢的情况处理模块，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的KeyThread.java。

```
if(!isnoPlay||isno_button_return) {                //按下暂停按钮，或者返回按钮
```

```
        continue;                              //不对任何物体进行模拟
    }
    time+=50;                                  //计时
    if(isnoVideo) {                            //如果是视频播放时
        PLANE_MOVE_SPAN=40;                    //飞机速度
        float nx,nz;                           //下一个目标点的坐标
    nx=ArchieArray[mapId][6][(playIdArray)*2]*WIDTH_LALNDFORM-PLANE_X;
    nz=ArchieArray[mapId][6][(playIdArray)*2+1]*WIDTH_LALNDFORM-PLANE_Z;
        float distance=(float) Math.sqrt(nx*nx+nz*nz);  //飞机到下一个点的距离
        if(nz<0) {                             //如果 nz 小于零
            rotationAngle_Plane_Y=DIRECTION_CAMERA-//计算其飞机飞行的朝向
                    (float)Math.toDegrees(Math.atan(nx/nz));
            }else if(nz==0) {                  //nz==0 时
                rotationAngle_Plane_Y=DIRECTION_CAMERA-nx>0?90:-90;
            }else {                            //否则根据 atan 值计算其飞行方向
                rotationAngle_Plane_Y=DIRECTION_CAMERA-
                    180+(float)Math.toDegrees(Math.atan(nx/nz));
            }
            rotationAngle_Plane_X=0;           //俯仰角度为 0
            ELEVATION_CAMERA=0;                //摄像机观察点距目标点的仰角
            rotationAngle_Plane_Z=0;           //z 方向的旋转角度为 0
            PLANE_Y=330;                       //飞机的高度
            if(distance<40) {                  //当飞机到该点的距离小于 40 时
                playIdArray++;                 //飞向下一个目标点
                playIdArray%=ArchieArray[mapId][6].length/2;
            }}
            if(gv.isGameMode==1&&!is_button_return&&!isVideo
            &&System.nanoTime()-gv.oldTime>1000000000){//特别行动时记录时间
            gv.goTime--;                       //总时间自减
            gv.oldTime=System.nanoTime();      //记录此时时间
            }
            if(gv.isGameMode==1) {             //特别行动
            if(gv.goTime<0) {                  //时间到了
                gv.isSpecActionState=2;        //标记行动失败
                if(gv.activity.bgMusic[1].isPlaying()){//如果背景音乐正在播放
                    gv.activity.bgMusic[1].pause();//暂停播放
                }
                isOvercome=true;               //游戏结束标志位，标志结束了
            }
            else if(gv.plane.blood<=0) {       //时间没有到——飞机炸毁了
                gv.isSpecActionState=2;        //标记行动失败
                if(gv.activity.bgMusic[1].isPlaying()){//如果正在播放背景音乐
                    gv.activity.bgMusic[1].pause();//暂停播放
                }
                isCrash=true;                  //飞机坠毁标志标记为已经坠毁
            }
            else{                              //时间还没有到，查看是否完成任务
                switch(mapId){
                case 3:                        //霹雳行动
```

```
                        if(enemy.size()==0) {      //如果敌机被全部歼灭
                            if(gv.activity.bgMusic[1].isPlaying()){
                                                   //如果正在播放背景音乐
                                gv.activity.bgMusic[1].pause();//暂停播放
                            }
                            isOvercome=true;  //游戏结束标志位标记为 true
                            gv.isSpecActionState=1; //行动成功
                        }
                        break;
                    case 4:                     //沙漠风暴，查看坦克和高射炮
                        if(tankeList.size()==0&&archie_List.size()==0) {
                                                //全部被歼灭
                            if(gv.activity.bgMusic[1].isPlaying()){
                                gv.activity.bgMusic[1].pause();
                            }
                            isOvercome=true;  //游戏结束标志位标记为结束
                            gv.isSpecActionState=1; //标记行动成功
                        }
                        break;
                    case 5:                     //斩首行动
                        if(arsenal.size()==0) {     //查看军火库是否被全部炸毁
                            if(gv.activity.bgMusic[1].isPlaying()){
                                gv.activity.bgMusic[1].pause();
                            }
                            isOvercome=true;  //游戏结束标记标记为已经结束
                            gv.isSpecActionState=1; //标记为行动成功
                        }
                        break;
                    }}}
```

> **说明**
>
> 场景预览是飞机绕着场景不断地循环飞行，此过程中没有绘制飞机，其原理是给出几个坐标点，然后飞机不断循环飞向下一个目标点。如果当前为特殊行动模式，则每次查看其是否行动成功，对应关卡查看对应物体是否被完全摧毁，即查看对应列表是否为空。

（3）接下来开发的是run方法中玩家飞机坠毁或者成功通关时的处理模块，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的KeyThread.java。

```
if(gv.isGameMode==0&&gv.plane.blood<=0) {                 //如果是战役模式且飞机生命值小于0
    if(gv.activity.bgMusic[1].isPlaying()){               //如果正在播放声音，则暂停播放
        gv.activity.bgMusic[1].pause();
    }
    isnoCrash=true;                                       //游戏是否结束，标志位结束
    isFireOn=false;                                       //是否发射子弹标记为不发射
    gradeArray[0]=mapId;                                  //记录此关的关卡编号
    gradeArray[2]=time/1000;                              //耗时分钟
    Date d=new Date();                                    //创建日期对象
        String month=d.getMonth()>=10?d.getMonth()+"":"0"+d.getMonth();//得到月份
```

```
        String day=d.getDate()>=10?d.getDate()+"":"0"+d.getDate();   //得到日
        String date=d.getYear()+1900+""+month+""+day;              //得到年
        String sql="insert into plane values("+"'"+gradeArray[0]+""+"'"//创建 SQL 语句
        +",'"+gradeArray[1]+""+"'," +"'"+gradeArray[2]+""+"','"+date+"');";
           SQLiteUtil.insert(sql);                                  //插入记录
        }
        if(arsenal.size()==0&&tankeList.size()==0&&archie_List.size()==0&&enemy.
        size()==0){
            if(gv.activity.bgMusic[1].isPlaying()){                  //暂停播放声音
                gv.activity.bgMusic[1].pause();
            }
            gradeArray[0]=mapId;                                     //记录关卡编号
            gradeArray[2]=time/1000;                                 //耗时分钟数
            Date d=new Date();                                       //创建时间对象
            String month=d.getMonth()>=10?d.getMonth()+"":"0"+d.getMonth();//得到月份
            String day=d.getDate()>=10?d.getDate()+"":"0"+d.getDate();//得到日
            String date=d.getYear()+1900+""+month+""+day;          //得到年
            String sql="insert into plane values("+"'"+gradeArray[0]+""+"'"+",'"+
            gradeArray[1]+""+"'," +
                    "'"+gradeArray[2]+""+"','"+date+"');"; //创建 SQL 语句
               SQLiteUtil.insert(sql);                          //插入数据
                isnoOvercome=true;                              //游戏结束标志设为 true
            }
            if(isnoCrash) {                                     //如果飞机坠毁
                isFireOn=false;
                archie_List.clear();                            //高射炮清零
                bomb_List.clear();                              //炮弹清零
                archie_bomb_List.clear();                       //高射炮炮弹清零
                bullet_List.clear();                            //飞机发射的子弹清零
                flag_go=false;                                  //关掉此线程
                new Thread(){                                   //创建线程模拟飞机大爆炸
                    int time=0;                                 //计时时间清零
                    int thistime=0;                             //当前时间
                    boolean isnoStart;                          //是否开始
                    public void run(){
                        isno_draw_arsenal=false;                //不绘制军火库
                        while(time<15000) {                     //时间小于 1.5s 时
                            time+=50;                           //时间自加
            planeY= isYachtHeadCollectionsWithLandPaodan        //计算飞机高度
                    (PLANE_X,PLANE_Y-PLANE_Y_R,PLANE_Z);//是否碰撞地面
                       if(planeY<0&&!isnoStart) {       //如果没有碰撞到地，飞机坠毁下降
            baoZhaList.add(new DrawBomb(bombRectr,PLANE_X,PLANE_Y,PLANE_Z));
            PLANE_Y-=10;                                  //飞机向下坠落
            tx=PLANE_X;                    //摄像机目标位置跟着飞机走，摄像机位置不变
            ty=PLANE_Y;
            tz=PLANE_Z;
            }else    {                     //如果已经坠毁到地面，拉伸摄像机，
            if(!isnoStart) {
             thistime=time;                //记录动画播放了的时间
```

```
            isnoStart=true;                                    //开始拉伸摄像机
            ty=PLANE_Y-isYachtHeadCollectionsWithLandPaodan//目标点为飞机位置
            (PLANE_X,PLANE_Y-PLANE_Y_R,PLANE_Z)+PLANE_Y_R;
            }}
            if(isnoStart&&time-thistime<=2000) {      //如果摄像机拉伸时间小于 2s
                ELEVATION_CAMERA=30;                  //摄像机方向仰角为 30°
    cx=(float)(tx+Math.cos(Math.toRadians(ELEVATION_CAMERA))*Math.sin(Math.
    toRadians
            (DIRECTION_CAMERA))*Crash_DISTANCE_start);  //摄像机 x 坐标
    cz=(float)(tz+Math.cos(Math.toRadians(ELEVATION_CAMERA)) *Math.cos(
            Math.toRadians(DIRECTION_CAMERA))*Crash_DISTANCE_start);//摄像机 z 坐标
    cy=(float)(ty+Math.sin(Math.toRadians(ELEVATION_CAMERA))
                *Crash_DISTANCE_start);               //摄像机 y 坐标
    if(Crash_DISTANCE_start<Crash_DISTANCE_stop*4){ //摄像机与飞机的距离到一定范围
                Crash_DISTANCE_start+=20f;            //距离不断增加
                otationAngle_Plane_Y=DIRECTION_CAMERA+=5;//摄像机围着飞机转动
        }else{
        rotationAngle_Plane_Y=DIRECTION_CAMERA+=5;  //摄像机围着飞机转动
        }
        if(time%800==0) {                           //每 800 ms 添加一个爆炸效果
        baoZhaList.add(new DrawBomb(bombRectr,PLANE_X,PLANE_Y,PLANE_Z));
        gv.activity.playSound(1,0);                 //并播放一个爆炸声音
        }}
        else if(isnoStart&&time-thistime>2000&&time-thistime<=5500)  {
                                                    //播放最后大爆炸
                if(time-thistime<4000) {            //如果在 4 s 内
                    if(time%150==0) {               //每 150 ms，让放大系数增大
                        BaoZha_scal+=0.3f;          //使爆炸矩形连续放大绘制
                    }}
        else if(time-thistime>=4000&&time-thistime<=4500) {  //如果是在 4s 到 4.5s 内
                    isno_draw_plane=false;          //不绘制飞机
                    isno_draw_arsenal=false;        //不绘制军火库
                    if(time%100==0){                //每 100ms 使其放大系数减少
                        BaoZha_scal-=3.5f;          //爆炸效果向内紧缩
                        if(BaoZha_scal<0){          //缩小到 0 时
        baoZhaList.add(new DrawBomb(bombRect,PLANE_X,PLANE_Y,PLANE_Z));
                BaoZha_scal=0;                      //系数为 0
        } } }else if(time-thistime>=4500){          //时间大于 4.5s 时
                    isnoCartoon=true;               //动画播放完毕
                    isno_draw_plane=false;          //不绘制飞机
                    gv.activity.playSound(0,1);     //播放爆炸效果声音
                    break;
                }}try {                             //线程睡眠 50ms
                    Thread.sleep(50);
                }catch (InterruptedException e)     {
                    e.printStackTrace();
                }}
        isnoCartoon=true;                           //动画播放完毕
            }}.start();
        break;
```

```
106             }
107             if(isnoOvercome) {                                    //如果完成游戏
108                 isFireOn=false;                                   //发射子弹为不发射
109                 flag_go=false;                                    //该线程结束
110                 archie_List.clear();                              //高射炮清零
111                 bomb_List.clear();                                //炮弹清零
112                 archie_bomb_List.clear();                         //高射炮炮弹清零
113                 bullet_List.clear();                              //飞机发射的子弹清零
114                 isnoCartoon=true;                                 //动画播放完毕
115                 break;
116             }
```

- 第1-16行为当飞机的生命值减少到0时，表示飞机坠毁，此时需要将其对应的数据放入数据库，然后在后面新建一个线程来模拟飞机坠毁的过程。
- 第17-31行为当玩家将所有敌方目标摧毁时，取得胜利，胜利后直接弹出对话框，玩家可选择进入下一关、重玩和菜单界面。
- 第32-106行为当玩家的飞机坠毁时，创建一个线程来模拟飞机坠毁和爆炸效果，其原理是，首先让飞机坠落到地面，然后围绕飞机旋转并且拉伸摄像机，最后通过连续放大绘制爆炸纹理矩形，最后使其瞬间缩小来模拟大爆炸效果。
- 第107-116行为在飞机坠毁或者玩家通关时，都要使该线程结束掉并清空对应的列表。

（4）接下来开发的是该类中run方法下飞机飞行的模拟模块，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的KeyThread.java。

```
if((!isPlaneNoUp)&&(keyState&0x1)!=0) {                    //如果有 up 键按下
    if(rotationAngle_Plane_X<PLANE_UP_ROTATION_DOMAIN_X){ //飞机向上仰
        rotationAngle_Plane_X+=PLANE_ROTATION_SPEED_SPAN_X;
    }
    if(ELEVATION_CAMERA>ELEVATION_CAMERA_DOWN) {           //摄像机向上仰
        ELEVATION_CAMERA-=ELEVATION_CAMERA_SPAN;           //角度减
    }}
    else if((keyState&0x2)!=0) {                           //如果有 down 键按下的话
    if(rotationAngle_Plane_X>PLANE_DOWN_ROTATION_DOMAIN_X){ //飞机向下俯
                rotationAngle_Plane_X-=PLANE_ROTATION_SPEED_SPAN_X;
        }
        if(ELEVATION_CAMERA<ELEVATION_CAMERA_UP){ //摄像机向下俯
            ELEVATION_CAMERA+=ELEVATION_CAMERA_SPAN;    //角度加
            }}
         else if((isPlaneNoUp||((keyState&0x1)==0)&&((keyState&0x2)==0))) {
            if(isno_Lock) {                                //锁定了敌方的某个目标
            if(!isno_adjust){                              //如果需要调整飞机飞行方向
            isno_adjust=true;                              //使飞机的飞行方向为对着锁定目标
            rotationAngle_Plane_Y=(float) Math.toDegrees(Math.atan(nx/nz));//
            if(nx==0&&nz==0){
            rotationAngle_Plane_Y=0;
            }if(nz>0) {
            rotationAngle_Plane_Y+=180;
            }
```

```
rotationAngle_Plane_X=(float) Math.toDegrees(Math.atan(ny/Math.sqrt
(nx*nx+nz*nz)));
DIRECTION_CAMERA=rotationAngle_Plane_Y;       //摄像机方向和飞机飞行方向一致
}}else{
    isno_adjust=false;                        //飞机的方向需要调整
if(Math.abs(rotationAngle_Plane_X)<           //飞机方向校正
    PLANE_ROTATION_SPEED_SPAN_X+0.1f) {       //如果飞机的俯仰角度小于一个阈值
    rotationAngle_Plane_X=0;                  //飞机前后持平
}else if(rotationAngle_Plane_X>0) {           //如果飞机向上仰,则角度减少
    rotationAngle_Plane_X-=PLANE_ROTATION_SPEED_SPAN_X;
}else{                                        //如果飞机下俯,则角度增加
    rotationAngle_Plane_X+=PLANE_ROTATION_SPEED_SPAN_X;
}
if(Math.abs(ELEVATION_CAMERA-ELEVATION_CAMERA_ORI) //摄像机校正
    <ELEVATION_CAMERA_SPAN+0.1f) {      //如果飞机的俯仰角度小于一个阈值
    ELEVATION_CAMERA=ELEVATION_CAMERA_ORI;//摄像机仰角矫正
}else if(ELEVATION_CAMERA-ELEVATION_CAMERA_ORI>0) {
    ELEVATION_CAMERA-=ELEVATION_CAMERA_SPAN;
}else{                                  //如果摄像机处于俯视的姿态,则角度增加
    ELEVATION_CAMERA+=ELEVATION_CAMERA_SPAN;
}}}
if(!isnoVideo){                         //不是视频播放界面，左右方向才有用
if((keyState&0x4)!=0) {                 //如果手机向左倾斜
float temp_ratio=Math.abs(gv.activity.directionDotXY[0]-gv.activity.
lr_domain)/30;
    DIRECTION_CAMERA+=DIRECTION_CAMERA_SPAN*temp_ratio; //摄像机左转
    rotationAngle_Plane_Y+=DIRECTION_CAMERA_SPAN*temp_ratio; //飞机向左旋转
    rotationAngle_Plane_Z=gv.activity.directionDotXY[0]*0.9f;
                                        //飞机视觉上向左旋转倾斜
}
else if((keyState&0x8)!=0){             //如果手机向右倾斜
float temp_ratio=Math.abs(gv.activity.directionDotXY[0]-gv.activity.lr_
domain)/30;
        DIRECTION_CAMERA-=DIRECTION_CAMERA_SPAN*temp_ratio;
        rotationAngle_Plane_Y-=DIRECTION_CAMERA_SPAN*temp_ratio;
        rotationAngle_Plane_Z=gv.activity.directionDotXY[0]*0.9f;
        }
else if((keyState&0x3)==0) {            //既没有向左又没有向右,飞机姿势摆正
    if(Math.abs(rotationAngle_Plane_Z)<PLANE_ROTATION_SPEED_SPAN_Z+0.5f) {
        rotationAngle_Plane_Z=0;            //飞机左右倾斜校正
    }else if(rotationAngle_Plane_Z<0){      //左倾斜摆正
    rotationAngle_Plane_Z+=PLANE_ROTATION_SPEED_SPAN_Z;
    }else    {
            rotationAngle_Plane_Z-=PLANE_ROTATION_SPEED_SPAN_Z;
        }
        if(Math.abs(rotationAngle_Plane_Y-DIRECTION_CAMERA)<2.1f) {
            rotationAngle_Plane_Y=DIRECTION_CAMERA;
        }                   //飞机视觉上 direction 与摄像机 direction 校正
        else if(rotationAngle_Plane_Y-DIRECTION_CAMERA<0) {
            rotationAngle_Plane_Y+=2;
        }else{
```

```
72                  rotationAngle_Plane_Y-=2;
73              }}}
74          planezAngle+=10;                          //敌机螺旋桨转动角度
75          planezAngle%=360;
76          PLANE_X-=Math.sin(Math.toRadians(rotationAngle_Plane_Y))* //计算飞机下一步位置
77                  Math.cos(Math.toRadians(rotationAngle_Plane_X))*PLANE_MOVE
                    _SPAN;
78          PLANE_Z-=Math.cos(Math.toRadians(rotationAngle_Plane_Y))*
79                  Math.cos(Math.toRadians(rotationAngle_Plane_X))*PLANE_MOVE
                    _SPAN;
80          float PLANE_Yt=(float) (PLANE_Y+Math.sin(Math.toRadians(rotationAngle_
            Plane_X))
81                  *PLANE_MOVE_SPAN);
```

- 第1-14行为有向上或者向下按钮按下时，调整飞机的俯仰角度，但其都有一个最大值，不能超过这个值。
- 第15-44行为在没有向上或者向下按钮按下时，如果飞机锁定了敌方的某一个目标，需要调整一下飞机的飞行方向，使其朝着目标飞行。然后使其慢慢地恢复到水平飞行状态。
- 第45-73行为当手机的姿态向左或者向右倾斜时并且不在场景预览界面，对其进行倾斜角度的调整，在场景预览界面时，玩家不能操控飞机的方向。
- 第74-81行为使螺旋桨旋转角度不断增加，使其旋转，然后根据飞机的方向和速度计算其下一个位置，而其y坐标需要在后面的碰撞检测中进行调整，所以此处不是直接给其赋值。

（5）接下来开发的是该类中run方法下的碰撞检测处理模块，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的KeyThread.java。

```
1    int gellSize=2;                                        //飞机飞离岛的地块数目最大值
2    if(PLANE_X<(-gellSize+0.5f)*WIDTH_LALNDFORM){    //左边不能离岛
3        PLANE_X=(-gellSize+0.5f)*WIDTH_LALNDFORM;
4    }
5    else if(PLANE_X>(MapArray[mapId].length+gellSize-0.5f)*WIDTH_LALNDFORM) {
6        PLANE_X=(MapArray[mapId].length+gellSize-0.5f)*WIDTH_LALNDFORM;
7    }
8    if(PLANE_Z<(-gellSize+0.5f)*WIDTH_LALNDFORM) {  //上边不能离岛
9        PLANE_Z=(-gellSize+0.5f)*WIDTH_LALNDFORM;
10       }
11       else if(PLANE_Z>(MapArray[mapId].length+gellSize-0.5f)*WATER_HEIGHT){
                                                              //下边不能离岛
12           PLANE_Z=(MapArray[mapId].length+gellSize-0.5f)*WATER_HEIGHT;
13       }try{
14       if(!isnoHitHill(PLANE_X,PLANE_Yt,PLANE_Z,          //碰撞检测
15               rotationAngle_Plane_Y,rotationAngle_Plane_X)){//如果没有和山地碰撞
16               PLANE_Y=PLANE_Yt;                            //对其赋值
17       }else{
18               if(!isnoVideo){                              //不是场景预览对其碰山后减生命值
19                   gv.plane.blood-=ArchieArray[mapId][9][4];    //飞机生命值减少
20                   gv.activity.playSound(0,0);              //播放飞机爆炸的声音
21                   gv.activity.shake();                     //手机振动一次
22               }}}catch(Exception e) {
23           e.printStackTrace();
24       }
25       if(PLANE_Y>=PLANE_HEIGHT_MAX){
26           isPlaneNoUp=false;                               //向上不管用
27           PLANE_Y=PLANE_HEIGHT_MAX;                        //飞机高度等于最大高度值
28       }else{
29           isPlaneNoUp=false;
30       }
31       rotationAngle_SkyBall+=0.1f;                         //天空穹旋转角度
32       RADAR_DIRECTION=rotationAngle_Plane_Y;               //这里计算雷达指针的旋转角度
33       tx=PLANE_X;                                          //摄像机的目标位置为飞机位置
34       ty=PLANE_Y;
35       tz=PLANE_Z;
36       cx=(float)(tx+Math.cos(Math.toRadians(ELEVATION_CAMERA))*     //摄像机x坐标
37           Math.sin(Math.toRadians(DIRECTION_CAMERA))*DISTANCE);
38       cz=(float)(tz+Math.cos(Math.toRadians(ELEVATION_CAMERA))*     //摄像机z坐标
39           Math.cos(Math.toRadians(DIRECTION_CAMERA))*DISTANCE);
40       cy=(float)(ty+Math.sin(Math.toRadians                          //摄像机y坐标
41                   (ELEVATION_CAMERA))*DISTANCE);
```

- 第1-13行为查看飞机是否飞出范围，将其调整到指定范围内。
- 第13-24行为对飞机进行碰撞检测，如果飞机碰到山，那么将飞机的高度调高到当前点的山的高度，并且自身的生命值减少一定值，如果没有碰到山则其高度为上面计算出来的高度。
- 第25-30行为查看飞机高度是否超过了指定高度，根据情况对其进行相应的调整。
- 第31-41行为使天空穹旋转，雷达指针的旋转角度为飞机的飞行方向，最后计算摄像机的位置。

（6）接下来开发的是该类中run方法下对其他各个物体的模拟模块，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的KeyThread.java。

```
gv.plane.arsenal_x=-scalMark*BUTTON_RADAR_BG_WIDTH*(//计算飞机在仪表板中的对应位置
        MapArray[mapId].length*WIDTH_LALNDFORM/2-tx)/
        (MapArray[mapId].length*WIDTH_LALNDFORM/2);
        gv.plane.arsenal_y=scalMark*BUTTON_RADAR_BG_WIDTH*
        (MapArray[mapId].length*WIDTH_LALNDFORM/2-tz)/
        (MapArray[mapId].length*WIDTH_LALNDFORM/2);
if(gv.plane.arsenal_x*gv.plane.arsenal_x+gv.plane.arsenal_y*gv.plane.arsenal_y
                                                                  //调整器位置
        >BUTTON_RADAR_BG_WIDTH*BUTTON_RADAR_BG_WIDTH*0.4f*0.4f){
        gv.plane.arsenal_x=(float) (gv.plane.arsenal_x*0.4f*
                (BUTTON_RADAR_BG_WIDTH/Math.sqrt(gv.plane.arsenal_x
                *gv.plane.arsenal_x+gv.plane.arsenal_y*gv.plane.arsenal_y)));
            gv.plane.arsenal_y=(float) (gv.plane.arsenal_y*0.4f*
                (BUTTON_RADAR_BG_WIDTH/Math.sqrt(gv.plane.arsenal_x*gv.plane.arsenal_x
                +gv.plane.arsenal_y*gv.plane.arsenal_y)));
        }try {
        Collections.sort(treeList);                               //对树排序
        } catch(Exception e) {e.printStackTrace();}
        for(Tree tree:treeList)   {                               //计算树的朝向
            try{
```

```
            tree.calculateBillboardDirection();      //排序
        }catch(Exception e){e.printStackTrace();}}
    if(WEAPON_INDEX==0&&isFireOn&&                    //进行发射子弹
        (System.nanoTime()-oldTimeBullet>150000000)) {
                                                      //这里设置连续发射子弹的时间间隔
        try{                                          //向列表中添加子弹对象
            if(bullet_number>0) {                     //发射子弹，添加时传入相关数据
    bullet_List.add(new BulletForControl(gv,gv.bullet_rect, PLANE_X, PLANE_Y, PLANE_Z,
            rotationAngle_Plane_X, rotationAngle_Plane_Y,rotationAngle_Plane_X,
            rotationAngle_Plane_Y,rotationAngle_Plane_Z,0,0));
    bullet_List.add(new BulletForControl(gv,gv.bullet_rect, PLANE_X, PLANE_Y, PLANE_Z,
            rotationAngle_Plane_X, rotationAngle_Plane_Y,rotationAngle_Plane_X,
            rotationAngle_Plane_Y,rotationAngle_Plane_Z,1,0));
                                                      //每次发射子弹为两边同时发射
                gv.activity.playSound(3,0);           //播放子弹的声音
                bullet_number-=2;                     //子弹数量减 2
                Collections.sort(bullet_List);        //对子弹排序
                }}catch(Exception ee) {ee.printStackTrace();}
            oldTimeBullet=System.nanoTime();          //记录这次发射子弹的时间
        }
    if(WEAPON_INDEX==1&&isFireOn&&                    //这里飞机进行发射炮弹
        (System.nanoTime()-oldTimeBomb>1000000000)) {//每隔一定时间才能发射一次炮弹
                    if(bomb_number>0) {
    try{                                              //向列表中添加炮弹对象
    gv.activity.shake();
    bomb_List.add(new BombForControl(gv,gv.bullet_ball, PLANE_X, PLANE_Y, PLANE_Z,
                rotationAngle_Plane_X, rotationAngle_Plane_Y,rotationAngle_Plane_X,
                rotationAngle_Plane_Y, rotationAngle_Plane_Z));
    }catch(Exception ee) {ee.printStackTrace();}
                        fire_index=(fire_index+1)%2;   //设定发射炮弹的机翼位置
                        gv.activity.playSound(4,0);    //播放声音
                        bomb_number--;                 //子弹数量减少
                        isno_Vibrate=true;
                        oldTimeBomb=System.nanoTime(); //记录此时发射子弹的时间
    }}
    if(!isnoVideo){                                   //是视频播放界面则不对其进行模拟
        for(int i=0;i<cop_bullet_List.size();i++){    //对子弹飞行进行模拟
         try{
            cop_bullet_List.get(i).go();
         }catch(Exception e) {e.printStackTrace();}}
        for(int i=0;i<cop_bomb_List.size();i++){      //对炮弹的飞行进行模拟
         try{
            cop_bomb_List.get(i).go();
         }catch(Exception e){e.printStackTrace();}}
        minimumdistance=Lock_Distance;            //锁定距离调到最大
    isno_Lock=false;                              //已经有物体被锁定，标志置为 false
    directionX=-(float) (Math.cos(Math.toRadians(rotationAngle_Plane_X))*
            Math.sin(Math.toRadians(rotationAngle_Plane_Y))); //飞机飞行的方向向量
    directionY=(float) (Math.sin(Math.toRadians(rotationAngle_Plane_X)));
    directionZ=-(float) (Math.cos(Math.toRadians(rotationAngle_Plane_X))*
            Math.cos(Math.toRadians(rotationAngle_Plane_Y)));
```

```
69        for(Arsenal_House ah:arsenal){                              //计算军火库
70            ah.calculateBillboardDirection();                        //查看军火库是否被锁定
71        }
72        for(int i=0;i<tankeList.size();i++){                        //对坦克进行模拟
73            try{
74                tankeList.get(i).tank_go();
75            }catch(Exception e){e.printStackTrace();}}
76        for(int i=0;i<cop_archie_List.size();i++){                  //对高射炮进行模拟
77            try{
78                cop_archie_List.get(i).go();
79            } catch(Exception e){e.printStackTrace();}}
80        for(int i=0;i<enemy.size();i++){                            //对敌机模拟
81            try{
82                enemy.get(i).go();
83            } catch(Exception e){e.printStackTrace();}}
84        for(int i=0;i<tank_bomb_List.size();i++){                   //对坦克炮弹进行模拟
85            try{
86                tank_bomb_List.get(i).go_tank();
87            } catch(Exception e){e.printStackTrace();}}
88        for(int i=0;i<cop_archie_bomb_List.size();i++){             //对高射炮炮弹进行模拟
89            try{
90                cop_archie_bomb_List.get(i).go_archie();
91            } catch(Exception e){e.printStackTrace();}
92        }}
```

- 第1-15行为根据飞机坐标计算飞机在仪表盘中的对应位置，当其对应位置超出仪表盘时，将其调整到边沿。
- 第22-52行为发射子弹和发射炮弹的处理，其中第22～37行为对发射子弹进行模拟，发射子弹为每次发射两颗，飞机的左右两边，设置其在一定时间间隔内发射一次，达到连续发射子弹的效果。第38-52行为发射炮弹，其每次只发射一颗，左右两边交替，而且需要较大的时间间隔才能发射一枚炮弹。
- 第53-92行为对其他各个物体进行模拟，主要是对像坦克等物体的发射炮弹的模拟，同时对敌方物体的锁定处理等，详细情况请见注释。

（7）接下来开发的是该类中的碰撞检测方法isnoHitHill，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的KeyThread.java。

```
1 public boolean isnoHitHill(float fjX,float fjY,float fjZ,float yAngle,float xAngle)
  {                                                      //碰撞检测方法
2         float fjLeftX,fjRightX;                        //飞机前面两个点的坐标
3         float fjLeftY,fjRightY;
4         float fjLeftZ,fjRightZ;                        //下面几行为计算飞机前左边点的坐标
5         fjLeftX=fjX-(float)(Math.cos(Math.toRadians(xAngle))*Math.sin(Math.
          toRadians(yAngle+ ANGLE_X_Z))*PLANE_X_R/(Math.sin(Math.toRadians
6                 (ANGLE_X_Z))));
7         fjLeftY=fjY-TRANSFER_Y-PLANE_Y_R+(float)(Math.sin(Math.toRadians(xAngle))
8                *PLANE_X_R/(Math.sin(Math.toRadians(ANGLE_X_Z))));
9         fjLeftZ=fjZ-(float)(Math.cos(Math.toRadians(xAngle))*Math.cos(Math.
          toRadians(yAngle+ANGLE_X_Z))*PLANE_X_R/(Math.sin(Math.toRadians
10                    (ANGLE_X_Z))));
11            if(isYachtHeadCollectionsWithLand(fjLeftX,fjLeftY,fjLeftZ)){
                                                         //调用碰撞检测方法查看此点
12                return true;
13            }                                          //下面几行为计算飞机右前方点的坐标
14            fjRightX=fjX-(float)(Math.cos(Math.toRadians(xAngle))*Math.sin(Math.
              toRadians(yAngle
15                    -ANGLE_X_Z))*PLANE_X_R/(Math.sin(Math.toRadians(ANGLE_X_Z))));;
16            fjRightY=fjY-TRANSFER_Y-PLANE_Y_R+(float)(Math.sin(Math.toRadians(xAngle))
17                    *PLANE_X_R/(Math.sin(Math.toRadians(ANGLE_X_Z))));
18            fjRightZ=fjZ-(float)(Math.cos(Math.toRadians(xAngle))*Math.cos(Math.
              toRadians(yAngle
19                    -ANGLE_X_Z))*PLANE_X_R/(Math.sin(Math.toRadians(ANGLE_X_Z))));
20            if(isYachtHeadCollectionsWithLand(fjRightX,fjRightY,fjRightZ)){
                                                         //调用碰撞检测方法查看此点
21                return true;
22            }
23            return false;
24        }
```

- 第5-12行为首先计算飞机左前方最低点的坐标，然后调用碰撞检测方法查看此点是否低于山地对应点的高度，如果是说明飞机和地面碰撞了。
- 第14-22行为计算飞机右前方最低点的坐标，然后调用碰撞检测方法，和左前方最低点一样。

说明

在此处，只察看飞机的前面左右方的最低点的高度是否低于地面对应点的高度来判断飞机是否和地面相碰撞，如果碰撞将飞机抬高到山地之上，即飞机的y坐标增加到山地此点的*y* 坐标。

（8）接下来开发的是该类中用于判断一点是否低于此点对应山地一点高度的方法isYachtHeadCollectionsWithLand，该方法根据传入

的点首先查看是否和军火库碰撞，然后是平房，最后是计算是否和山地碰撞，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的KeyThread.java。

```
public  boolean isYachtHeadCollectionsWithLand(float ctx,float ctyh,float ctz){
        for(Arsenal_House as:arsenal){ //判断其是否撞军火库了
            if(ctyh>as.ty&&ctyh<as.ty+ARSENAL_Y              //与军火库相撞了
                    &&ctx>as.tx-ARSENAL_X&&ctx<as.tx+ARSENAL_X
                    &&ctz>as.tz-ARSENAL_Z&&ctz<as.tz+ARSENAL_Z){
                gv.plane.blood-=ArchieArray[mapId][9][3];        //播放声音
                PLANE_Y=as.ty+ARSENAL_Y;             //将飞机高度抬高到大于军火库的高度
                isno_Hit=true;                       //碰撞标志位置为true
                return true;                         //返回已经碰撞状态true
                }}
            for(int i=0;i<ArchieArray[mapId][4].length/2;i++){   //判断是否与平民房相碰
            float positionX=ArchieArray[mapId][4][2*i]*WIDTH_LALNDFORM;//,LAND_HIGHEST
                        +house_height/2, ArchieArray[mapId][4][2*i+1]*WIDTH_LALNDFORM
            float positionZ=ArchieArray[mapId][4][2*i+1]*WIDTH_LALNDFORM;
                                                     //计算得到平房的位置
            if (ctx>(positionX-house_length/2)&&ctx<(positionX+house_length/2)&&
                                                     //查看其是否碰撞
```

```
            ctz>(positionZ-house_width/2)&&ctz<(positionZ+house_width/2)&&
            ctyh>LAND_HIGHEST&&ctyh<(LAND_HIGHEST+house_height)){
        gv.plane.blood-=ArchieArray[mapId][9][4];   //如果碰撞则播放声音
        PLANE_Y+=house_height;                        //将飞机高度抬高到平房之上
        isno_Hit=true;                                //碰撞标志位置 true
        return true;
    }}
    float UNIT_SIZE=LAND_UNIT_SIZE;                   //陆地每一个小格子的大小
    int COLS=LANDS_HEIGHT_ARRAY[0].length;            //行数和列数
    int ROWS=LANDS_HEIGHT_ARRAY[0].length;
    float cellCount=0;                                //需要移动的格子数
    float smallBlockLength=WIDTH_LALNDFORM;//UNIT_SIZE*COLS;//每一个地块的大小
    ctx=ctx+cellCount*smallBlockLength;               //将地图移动到都大于零的区域
    ctz=ctz+cellCount*smallBlockLength;
    int col=(int)(ctx/smallBlockLength);              //飞机所在的地块的行列
    int row=(int)(ctz/smallBlockLength);              //下面为查看飞机如果在岛外
    if(col<0||row<0||col>MapArray[mapId].length-1||row>MapArray[mapId].
    length-1) {
        if(ctyh<0){                                   //如果飞机低于水面，则碰撞了
            PLANE_Y+=0-ctyh;                          //将飞机抬高到水面之上
            return true;
        }else{
            return false;
        }}
    ctx=ctx-col*smallBlockLength;                     //将该点移动到相对于原点的坐标处
    ctz=ctz-row*smallBlockLength;
    int mapArrayId=MapArray[mapId][row][col];        //得到对应块的编号
    float moderXZ=0;                                  //中间变量
    switch (mapArrayId) {
    case 4:                                       //4 号地块由 0 号地块旋转 90°得到
        moderXZ=ctz;                              //z 坐标等于 x 坐标，x 坐标为 z 坐标翻转
        ctz=ctx;
        ctx=smallBlockLength-moderXZ;
        mapArrayId=0;
        break;
    …//此处省略部分代码，读者可自行查阅随书光盘中的源代码
    case 20:                                          //竖着山的下面山地块
        moderXZ=ctz;
        ctz=smallBlockLength-ctx;
        ctx=moderXZ;
        mapArrayId=5;
        break;
    default:
        break;
    }
    try{
       int tempCol=(int)(ctx/UNIT_SIZE)%7;            //得到所在地块中的行列
       int tempRow=(int)(ctz/UNIT_SIZE)%7;
       float yArray[][]=Constant.LANDS_HEIGHT_ARRAY[mapArrayId];
       float x0=tempCol*UNIT_SIZE;                    //得到对应行列格子的左上方坐标
       float z0=tempRow*UNIT_SIZE;
```

```
66                  float y0=yArray[tempRow][tempCol];
67                  float x1=x0+UNIT_SIZE;             //得到格子右上方顶点处的坐标
68                  float z1=z0;
69                  float y1=yArray[tempRow][(tempCol+1)%COLS];
70                  float x2=x0+UNIT_SIZE;             //得到格子右下方顶点处的坐标
71                 float z2=z0+UNIT_SIZE;
72                 float y2=yArray[(tempRow+1)%ROWS][(tempCol+1)%COLS];
73                  float x3=x0;                       //得到格子左下方顶点的坐标
74                  float z3=z0+UNIT_SIZE;
75                  float y3=yArray[(tempRow+1)%ROWS][(tempCol)%COLS];
76                  float cty=0;
77                  if(isInTriangle(x0,z0,x1,z1,x3,z3,ctx,ctz)) {
                                                       //判断该点是否位于0-1-3三角形
78                   cty=fromXZToY(                    //求0-1-3面在该点处的高度
79                           x0,y0,z0, x1,y1,z1,x3,y3,z3,ctx,ctz); }
80                  else if(isInTriangle(x2,z2,x3,z3,x1,z1,ctx,ctz)) {
81                   cty=fromXZToY(                    //求1-2-3面在该点处的高度
82                           x1,y1,z1, x2,y2,z2, x3,y3,z3,ctx,ctz);
83                  }
84                  if(cty>ctyh){                      //若飞机处的陆地低于飞机高度则返回true
85                   PLANE_Y+=cty-ctyh;
86                   return true;
87                  }
88              }
89              catch(Exception e){ e.printStackTrace();}        //异常处理
90             return false;
91         }
```

• 第2-22行为查看其是否与军火库或平房相碰撞了，其原理为根据其位置，然后设立一个包围盒，如果该点在该包围和内，则说明碰撞了。

• 第23-41行为首先根据其坐标计算得到所在行列，然后将该点对应到原点处地块上，便于计算，再判断其是否在岛外，如果在岛外，直接查看其是否低于水面即可，最后根据其原来所处行列得到对应的地块编号。

• 第43-59行为根据此编号，来计算其对应到原点处地块上的坐标和对应使用的基本地块。

• 第60-76行为根据计算得到的坐标计算所在的地块中的行列，然后计算此小块的4个角的坐标，每个点的高度从对应地块的高度数组中取得。

• 第77-83行为传入此地块中一个三角形3个点的坐标来查看该点是否在某个三角形内，然后返回对应此三角形此点的高度。

• 第84-88行查看其是否碰撞，即此点高度是否低于此点对应山地处的高度。

说明

游戏的主要逻辑处理方法基本介绍完毕，对于后面查看一点是否在某个三角形内和计算此点对应此三角形处高度的方法，此处就不再对其进行介绍了，请读者自行查阅随书光盘中的源代码。

22.5.3　子弹类BulletForControl

本节将对子弹类BulletForControl类进行介绍，该类对子弹的发射、飞行和击中目标等进行模 拟，子弹使用标志板技术，用一个纹理矩形模拟子弹，每次绘制时使该矩形垂直于子弹位置到摄像机位置的方向绘制，开发该类的详细步骤如下。

（1）首先开发的是BulletForControl类的框架，具体代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的BulletForControl.java。

```
package com.bn.gameView;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘中的源代码
public class BulletForControl implements Comparable<BulletForControl>{//子弹类
    private TextureRect bullet_rect;                           //代表子弹的纹理矩形
    private float curr_x;                                      //发射子弹时的位置
    private float curr_y;
    private float curr_z;
    private float curr_elevation;                              //发射子弹时的仰角和方位角
    private float curr_direction;
        private float distance;                                //子弹飞行距离
        private boolean islocked;
        private float curr_nx;                                 //锁定状态下的子弹飞行的方向向量
        private float curr_ny;
        private float curr_nz;
        private float average;                                 //平均向量
        private float curr_rotation;                           //子弹的旋转角度
        GLGameView gv;
        public int bulletId;  //子弹的 ID，看是敌机发出的还是自己发出的子弹，1 为敌机发出的子弹
        public BulletForControl(GLGameView gv,TextureRect bullet_rect,//构造器
            float plane_x,float plane_y,float plane_z,float plane_elevation,float
            plane_direction
            ,float rotationAngle_Plane_X,float rotationAngle_Plane_Y, float
            rotationAngle_Plane_Z,
            int bulletIndex,int bulletId) {//bulletIndex 表示是左机翼发射子弹还是右机翼
            this.bulletId=bulletId;
            this.gv=gv;
            this.bullet_rect=bullet_rect;                      //创建纹理矩形引用
            this.curr_x=plane_x;                               //起始位置
            this.curr_y=plane_y;
            this.curr_z=plane_z;
            this.curr_elevation=plane_elevation;               //起始角度
            this.curr_direction=plane_direction;
            this.islocked=isno_Lock;                           //是否有锁定目标
            if(islocked) {                                     //如果有锁定
                curr_nx=nx;                                    //则发射子弹方向为朝向目标发射
                curr_ny=ny;
                curr_nz=nz;
                average=(float) Math.sqrt(curr_nx*curr_nx+curr_ny*curr_ny+curr_nz*
                curr_nz);
            }
            initData(plane_x,plane_y,plane_z,rotationAngle_Plane_X,
                                                               //计算子弹的初始位置方法
                rotationAngle_Plane_Y,rotationAngle_Plane_Z,bulletIndex);
        }
```

```
43        public void initData(float plane_x,float plane_y,float plane_z,float
          rotationAngle_Plane_X,
44                 float rotationAngle_Plane_Y,float rotationAngle_Plane_Z,int bulletIndex){
45        ……//此处省略了确定飞机机翼子弹的发射位置的部分代码，在后面的步骤中给出
46        }
47        public void drawSelf(int texId) {                          //子弹绘制方法
48        ……//此处省略了纹理矩形的绘制的部分代码，将在后面的步骤中给出
49        }
50        public void go(){                                          //子弹飞行一步方法
51        ……//此处省略了子弹前进一步方法中的部分代码，将在后面的步骤中给出
52        }
53        public void calculateBillboardDirection(){                 //这里计算标志板的朝向
54        ……//此处省略了计算纹理矩形朝向的部分代码，将在后面的步骤中给出
55        }
56        @Override
57        public int compareTo(BulletForControl another) {
                                              //重写的比较两个粒子离摄像机距离的比较方法
58        ……//此处省略了排序时调用的方法中的部分代码，将在后面的步骤中给出
59        }}
```

- 第43-46行为计算飞机机翼发射子弹的发射位置，飞机每次发射子弹为两边同时发射，所以需要计算其对应位置。
- 第50-52行为飞机飞行一步的方法，其每飞行一步，根据其属于那方子弹来做对应的计算，如果是敌方子弹，则查看其是否击中玩家的飞机，否则查看其是否击中敌方的某个物体，并作出对应的操作，最后需要计算其朝向，使其始终垂直于摄像机。

（2）接下来开发的是该类中的计算子弹初始位置的方法initData，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的BulletForControl.java。

```
public void initData(float plane_x,float plane_y,float plane_z,float rotationAngle_Plane_X,
        float rotationAngle_Plane_Y,float rotationAngle_Plane_Z,int bulletIndex) {
        curr_x=plane_x;                                    //得到飞机位置
        curr_y=plane_y;
        curr_z=plane_z;
        float length;                                      //炮弹位置的相关参数
        float ori_y;
        float ori_z;
        length=6;
            if(bulletIndex!=1) {                           //左机翼发射子弹
                ori_y=90;                                  //调整值
                ori_z=-2;
            }else{                                         //右机翼发射子弹
                ori_y=-90;                                 //调整值
                ori_z=-2;
            }                                              //根据飞机位置和调整值计算最终位置
            curr_y=curr_y-(float)Math.sin(Math.toRadians(rotationAngle_Plane_Z+ori_
            z))*length;
            curr_x=curr_x-(float)Math.cos(Math.toRadians(rotationAngle_Plane_Z+ori_z))*
                        (float)Math.sin(Math.toRadians(rotationAngle_Plane_Y+ori
                              _y))*length;
            curr_z=curr_z-(float)Math.cos(Math.toRadians(rotationAngle_Plane_Z+ori_z))*
                        (float)Math.cos(Math.toRadians(rotationAngle_Plane_Y+ori
                        _y))*length;
        }
```

说明

该方法的作用为根据此子弹是左边还是右边发射的，来标记计算其子弹的初始位置。

（3）接下来开发的是该类中模拟子弹飞行的go方法，该方法首先判断子弹属于哪一方，然后查看其是否击中某个物体，最后根据情况做出相应的处理，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/gameView目录下的BulletForControl.java。

```java
public void go(){
    distance+=BULLET_VELOCITY;                              //子弹的行驶距离增加
    if(distance>=BULLET_MAX_DISTANCE) {                     //如果子弹步长超出了
        Iterator<BulletForControl> ite=bullet_List.iterator();   //从列表中移除该子弹
        while(ite.hasNext()){
            if(ite.next()==this){
                ite.remove();
                return;
            }}}
        if(bulletId==0) {                                        //如果不是敌方子弹
        for(Arsenal_House as:arsenal) {                          //判断是否和军火库发生碰撞
            if(curr_y>as.ty&&curr_y<as.ty+ARSENAL_Y              //军火库还存在
                &&curr_x>as.tx-ARSENAL_X&&curr_x<as.tx+ARSENAL_X
                &&curr_z>as.tz-ARSENAL_Z&&curr_z<as.tz+ARSENAL_Z) {
                as.blood-=ArchieArray[mapId][7][0];              //军火库的生命值减少
                if(as.blood<1){                                  //如果军火库被炸毁
                    gv.activity.playSound(0,0);                  //播放爆炸声音
                    try{
                        Iterator<Arsenal_House> ite=arsenal.iterator();
                        while(ite.hasNext()){                    //从对应列表中移除该军火库
                            if(ite.next()==as) {
                                ite.remove();
                        }}}catch(Exception e) {e.printStackTrace();}
                    gradeArray[1]+=ArchieArray[mapId][12][3];    //得分增加
                    baoZhaList.add(new DrawBomb(bombRect,as.tx,as.ty+bomb_height/2,
                    as.tz));
                    }
                    Iterator<BulletForControl> ite=bullet_List.iterator();
                    while(ite.hasNext()){                        //将该子弹从列表中删除
                        if(ite.next()==this) {
                            ite.remove();
                            return;
                        }}}}
        for(ArchieForControl afc:archie_List) {                  //查看有没有击中高射炮
            if(curr_y>afc.position[1]&&curr_y<afc.position[1]+ARCHIBALD_Y
            &&curr_x>afc.position[0]-ARCHIBALD_X&&curr_x<afc.position[0]+ARCHIBALD_X
```

```
            &&curr_z>afc.position[2]-ARCHIBALD_Z&&curr_z<afc.position[2]+ARCHIBALD_Z){
            afc.blood-=ArchieArray[mapId][7][2];          //高射炮的血减1
            try{
                if(afc.blood<1) {                         //如果高射炮炸毁
                        gv.activity.playSound(0,1);      //播放爆炸声音
                        try{
                            Iterator<ArchieForControl> ite=archie_List.iterator();
                            while(ite.hasNext()){        //将该高射炮从对应列表中移除
                                if(ite.next()==afc) {
                                    ite.remove();
                                }}} catch(Exception e) {e.printStackTrace();}
            gradeArray[1]+=ArchieArray[mapId][12][1];    //得分增加
            baoZhaList.add(new DrawBomb(bombRect,curr_x,curr_y+bomb_height/2,curr_z));
            }} catch(Exception e) {e.printStackTrace();}
            Iterator<BulletForControl> ite=bullet_List.iterator();//将该子弹从列表中移除
            while(ite.hasNext()){
                    if(ite.next()==this) {
                        ite.remove();
                        return;
                    }}}}
        for(TanKe afc:tankeList){                         //查看有没有击中坦克
            if(curr_y>afc.ty&&curr_y<afc.ty+ARCHIBALD_Y
                    &&curr_x>afc.tx-ARCHIBALD_X&&curr_x<afc.tx+ARCHIBALD_X
                    &&curr_z>afc.tz-ARCHIBALD_Z&&curr_z<afc.tz+ARCHIBALD_Z){
                afc.blood-=ArchieArray[mapId][7][1];     //坦克的血减1
                if(afc.blood<=0) {                        //如果坦克被炸毁
                    gv.activity.playSound(0,1);
                    try{
                        Iterator<TanKe> ite=tankeList.iterator();
                        while(ite.hasNext()){             //将该坦克从对应列表中移除
                            if(ite.next()==afc) {
                                ite.remove();
                            }}} catch(Exception e) {e.printStackTrace();}
            baoZhaList.add(new DrawBomb(bombRect,curr_x,curr_y+bomb_height/2,curr_z));
            gradeArray[1]+=ArchieArray[mapId][12][0];    //得分增加
            }
            Iterator<BulletForControl> ite=bullet_List.iterator();
            while(ite.hasNext()){                         //将该子弹从对应列表中移除
                            if(ite.next()==this) {
                                ite.remove();
                                return;
                            }}}}
        for(EnemyPlane afc:enemy){                        //有没有击中敌机
            if(curr_y>afc.ty-PLANE_Y_R&&curr_y<afc.ty+PLANE_Y_R
                    &&curr_x>afc.tx-PLANE_X_R&&curr_x<afc.tx+PLANE_X_R
                    &&curr_z>afc.tz-ANGLE_X_Z&&curr_z<afc.tz+ANGLE_X_Z){
                afc.blood-=ArchieArray[mapId][7][3];     //敌机的血减少
                if(afc.blood<=0){                         //如果敌机被炸毁
                    gv.activity.playSound(0,1);          //播放爆炸声音
                    try{
                        Iterator<EnemyPlane> ite=enemy.iterator();
                        while(ite.hasNext()){             //将该敌机从对应列表中移除
```

```
88                          if(ite.next()==afc) {
89                              ite.remove();
90                          }}} catch(Exception e) {e.printStackTrace();}
91              baoZhaList.add(new DrawBomb(bombRect,curr_x,curr_y+bomb_height/5,curr_z));
92              gradeArray[1]+=ArchieArray[mapId][12][2];        //得分增加
93              }
94              Iterator<BulletForControl> ite=bullet_List.iterator();
95              while(ite.hasNext()){                              //将子弹从对应列表中移除
96                      if(ite.next()==this) {
97                          ite.remove();
98                          return;
99                      }}}}}
100             else{                                              //如果是敌方的子弹
101                 float curr_planeX=PLANE_X;
102                 float curr_planeY=PLANE_Y;
103                 float curr_planeZ=PLANE_Z;
104                 float curr_distance=(curr_planeX-curr_x)*(curr_planeX-curr_x)+
105                 (curr_planeY-curr_y)*(curr_planeY-curr_y)+
106                 (curr_planeZ-curr_z)*(curr_planeZ-curr_z);
107                 if(curr_distance<500) {                        //炮弹与飞机相撞
108                     gv.plane.blood-=ArchieArray[mapId][9][2];   //飞机血减少
109                     try{
110                         Iterator<BulletForControl> ite=bullet_List.iterator();
111                         while(ite.hasNext()){                  //将该子弹从对应列表中移除
112                             if(ite.next()==this) {
113                                 ite.remove();
114                                 return;
115                     }}} catch(Exception e) {e.printStackTrace();}
116                     return ;
117                 }}
118             if(islocked) {                        //如果锁定目标，则子弹按照锁定方向飞行
119                  curr_x+=curr_nx/average*BULLET_VELOCITY;
120                  curr_z+=curr_nz/average*BULLET_VELOCITY;
121                  curr_y+=curr_ny/average*BULLET_VELOCITY;
122             }else{
123         curr_x=curr_x-(float)(Math.cos(Math.toRadians(curr_elevation))* //计算子弹位置
124                 Math.sin(Math.toRadians(curr_direction))*BULLET_VELOCITY);
125         curr_z=curr_z-(float)(Math.cos(Math.toRadians(curr_elevation))*
126                 Math.cos(Math.toRadians(curr_direction))*BULLET_VELOCITY);
127         curr_y=curr_y+(float)(Math.sin(Math.toRadians(curr_elevation))*BULLET_VELOCITY);
128         }
129         calculateBillboardDirection();                //计算朝向
130         }
```

- 第2-9行为每次将子弹飞行距离增加，当达到一定的值时，此子弹将不再有效，将其从子弹列表中移除即可。
- 第10-99行为当此子弹不为敌方子弹时，需要查看其是否击中敌方的军火库、高射炮、坦克和飞机，当击中某一个物体时，首先查看其是否被击毁，如果击毁则将该物体从对应列表中移除，然后将该子弹从列表中移除。
- 第100-117行为当此子弹为敌方发射的子弹时，计算该子弹与飞机的距离，如果小于某个值则说明其子弹击中了飞机。
- 第118-128行为根据当前此子弹在发射时是否为锁定状态，如果是则根据锁定状态时的发射方向飞行，否则根据其原来方向飞行。

- 第129行为计算此子弹的朝向，使该子弹每次垂直于摄像机绘制。

> **说明**
>
> 对该类中的其他方法此处就不再做过多的介绍。而对于炮弹类，其原理和子弹类没有太大的区别，不同的是它们击中对方时，该物体的生命值减少的不同些，炮弹不用计算绘制朝向，所以本章就不再对其进行详细介绍了，请读者自行查看随书光盘中的源代码。

22.5.4　坦克类TanKe

本节对坦克类的开发进行介绍，该类包括对坦克模型的绘制、坦克在仪表盘上对应标识的绘制和坦克发射炮弹的模拟等。开发该类的详细步骤如下。

（1）首先开发的是TanKe类的框架，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/tankemodel目录下的TanKe.java。

```
package com.bn.tankemodel;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘中的源代码
public class TanKe {
    Model tanke_body, tanke_barrel;                    //模型
    private BallTextureByVertex bomb_ball;             //炮弹球
    GLGameView gv;
    public float tx,ty,tz;
    public float[] tank_position;                      //坦克的位置
    public float barrel_bottom=16;                     //炮管底部的高度
        public float[] barrel_bottom_position=new float[3]; //坦克炮管底部的位置
        public float barrel_length=30;                 //炮管的长度
        int row;                                       //所在行列
        int col;
        float tank_barrel_direction=0;                 //坦克炮管的方向角
        float tank_barrel_elevation=0;                 //坦克炮管的仰角
        public float[] bomb_position_init=new float[3]; //坦克发射炮弹的初始位置
        private float oldTime=0;                       //用于记录炮弹上次发射的时间
        public NumberForDraw nm;                       //数字引用
        public TextureRect backgroundRect;             //背景
        public float xue_scale=0.6f;                   //血背景缩放比例
        public int blood=100;                          //坦克的生命值
        int drawblood;                                 //绘制时用于复制坦克生命值的变量
        float yAnglexue;                               //坦克炮管的朝向
        public boolean this_isno_Lock=false;       //自己是否被锁定，被锁定就绘制锁定矩形
        public TextureRect mark_lock;                  //标记被锁定的矩形
        public TextureRect mark_plane;                 //在仪表盘上标记的矩形
        float arsenal_x,arsenal_y,arsenal_z;           //坦克在仪表盘上的位置坐标
        public TanKe(GLGameView gv,BallTextureByVertex bomb_ball,Model tanke_body,
                    Model tanke_barrel,float[] tank_position,int col,int row,
```

```
30                  TextureRect backgroundRect,NumberForDraw nm,
31                  TextureRect mark_plane,TextureRect mark_lock){
32          this.nm=nm;                                        //数字
33          this.backgroundRect=backgroundRect;                //血背景
34          this.gv=gv;                                        //
35          this.bomb_ball=bomb_ball;                          //炮弹
36          this.tanke_body=tanke_body;                        //坦克身体模型
37          this.tanke_barrel=tanke_barrel;                    //坦克炮管
38          this.tank_position=tank_position;                  //初始化坦克的位置
39          this.barrel_bottom_position[0]=tank_position[0];
40          this.barrel_bottom_position[1]=tank_position[1]+barrel_bottom;
41          this.barrel_bottom_position[2]=tank_position[2];
42          this.col=col;                                      //所在的地块格子
43          this.row=row;
44          tx=tank_position[0];                               //坦克位置
45          ty=tank_position[1];
46          tz=tank_position[2];
47          this.mark_plane=mark_plane;                        //标记矩形
48          arsenal_x=-scalMark*BUTTON_RADAR_BG_WIDTH*(MapArray[mapId].length
49                  *WIDTH_LALNDFORM/2-tx)/(MapArray[mapId].length*WIDTH_LALNDFORM/2);
50          arsenal_y=scalMark*BUTTON_RADAR_BG_WIDTH*(MapArray[mapId].length*
51                  WIDTH_LALNDFORM/2-tz)/(MapArray[mapId].length*WIDTH_LALNDFORM/2);
52          this.mark_lock=mark_lock;                          //锁定矩形
53      }
54      public void drawSelf(int texId,int ii,int jj,int rowR,int colR,
55              int backgroundRectId,int numberID,int locktexId){
56      ……//此处省略了绘制坦克方法中的部分代码，将在后面的步骤中给出
57      }
58      public void drawSelfMark(int texId){
59      ……//此处省略了绘制仪表盘上标记方法中的部分代码，将在后面的步骤中给出
60      }
61      public void calculateBillboardDirection(){             //计算生命值矩形的朝向
62      ……//此处省略了查看是否被锁定方法中的部分代码，将在后面的步骤中给出
63      }
64      public void tank_go() {                                //时时改变坦克炮管的角度
65      ……//此处省略了时时改变坦克炮管的角度方法中的部分代码，将在后面的步骤中给出
66      }}
```

- 第28-53行为构造器，此构造器需要一次传入坦克模型、位置、表示生命值的矩形引用、锁定框引用和标识位置矩形引用等，然后根据其在3D场景中的位置计算其对应仪表盘上的位置。
- 第54-57行为相关物体的绘制方法，包括坦克模型、坦克上方的表示坦克生命值的矩形和在被飞机锁定情况下的锁定框等。
- 第58-60行为绘制仪表盘上标识坦克的，在对应位置的矩形。
- 第61-63行为计算表示生命值矩形的朝向和查看其是否被锁定。
- 第64-66行为计算坦克炮管的朝向，在条件满足的情况下发射炮弹。

（2）接下来开发的是该类中的坦克绘制方法drawSelf，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/tankemodel目录下的TanKe.java。

```
1 public void drawSelf(int texId,int ii,int jj,int rowR,int colR,
2         int backgroundRectId,int numberID,int locktexId){
3         if(row<ii||row>rowR||col<jj||col>colR){          //如果坦克不在该绘制的格子内
4             return;                                      //直接返回不绘制坦克
5         }
6         drawblood=blood;                                 //复制当前生命值
7         MatrixState.pushMatrix();                        //保护现场
8         MatrixState.translate(tank_position[0], tank_position[1], tank_position[2]);
9         MatrixState.scale(tank_ratio, tank_ratio, tank_ratio);   //缩放
10            tanke_body.drawSelf(texId);                          //绘制坦克身体
11            MatrixState.translate(0, barrel_bottom, 0);          //推动坐标到炮管位置
12            MatrixState.rotate(tank_barrel_direction, 0, 1, 0);//绕y轴旋转使其对准飞机
13            MatrixState.rotate(tank_barrel_elevation, 1, 0, 0);//绕x轴旋转使其对准飞机
14            tanke_barrel.drawSelf(texId);        //绘制炮管
15            MatrixState.popMatrix();             //恢复现场
16            GLES20.glEnable(GLES20.GL_BLEND);    //开启混合
17            GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA, GLES20.GL_ONE_MINUS_SRC_ALPHA);
18            if(drawblood>=0){                    //生命值大于0，绘制代表生命值的矩形
19                MatrixState.pushMatrix();        //保护现场
20                MatrixState.translate(tank_position[0], tank_position[1]+70, tank_
                  position[2]);
21                MatrixState.scale(xue_scale, xue_scale, xue_scale); //缩放
22                MatrixState.rotate(yAnglexue, 0,1, 0);      //绕y轴旋转对准摄像机
23                backgroundRect.bloodValue=drawblood*2-100+6;//将生命值对应到矩形坐标
24                backgroundRect.drawSelf(backgroundRectId);  //绘制代表生命值的矩形
25                MatrixState.popMatrix();                    //恢复现场
26            }
27            if(this.isno_Lock){                             //如果是自己被锁定绘制锁定框
28                MatrixState.pushMatrix();                   //保护矩阵
29                MatrixState.translate(tank_position[0], tank_position[1]+20, tank_
                  position[2]);
30                MatrixState.rotate(yAnglexue, 0,1, 0); //绕y轴旋转对着摄像机方向
31                MatrixState.rotate(rotationAngle_Plane_Z, 0,0, 1);   //绕z轴旋转
32                MatrixState.scale(tank_ratio, tank_ratio, tank_ratio);   //缩放
33                mark_lock.drawSelf(locktexId);              //绘制
34                MatrixState.popMatrix();                    //恢复现场
35            }
36            GLES20.glDisable(GLES20.GL_BLEND);              //关闭混合
37    }
```

- 第3-15行，首先判断该坦克是否在绘制的地块范围内，是则绘制坦克，首先绘制坦克的下面部分，然后根据炮管的朝向绘制坦克的炮管。
- 第18-26行为该坦克的生命值大于0的情况下，绘制表示坦克生命值的矩形。

- 第27-35行为如果此坦克被锁定了，则绘制其在锁定状态下的锁定框。

（3）接下来开发的是该类中计算生命值矩形朝向和该坦克是否被锁定的方法calculateBillboard Direction，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/tankemodel目录下的TanKe.java。

```
1 public void calculateBillboardDirection(){           //计算代表生命值的矩形框的朝向
2         float currX_span=tx-cx;                      //得到飞机与坦克直接的对应轴的向量
3         float currZ_span=tz-cz;
4         if(currZ_span<0){                            //使用正切值计算其朝向角度
5             yAnglexue=(float)Math.toDegrees(Math.atan(currX_span/currZ_span));
6         }else if(currZ_span==0){
7             yAnglexue=currX_span>0?90:-90;
8         }else {
9             yAnglexue=180+(float)Math.toDegrees(Math.atan(currX_span/currZ_span));
10            }
11            if(isno_Lock) {                          //如果已经有被锁定的目标了
12                this_isno_Lock=false;
13                return;
14            }
15            float x1,y1,z1,x2,y2,z2;                 //计算其是否被锁定
16            x1=tx-PLANE_X;                           //得到对应轴的距离
17            y1=ty-PLANE_Y;
18            z1=tz-PLANE_Z;
19            float distance1=(float) Math.sqrt(x1*x1+y1*y1+z1*z1);    //得到距离
20            if(distance1>minimumdistance){//如果距离超出范围，或者已经有一个被锁定了，则返回
21                this_isno_Lock=false;                //锁定标志位为不被锁定
22                return;
23            }                                        //否则说明自己可能被锁定了
24            x2=directionX;                           //计算飞机飞行的方向向量
26            y2=directionY;
27            z2=directionZ;                   //计算飞机飞行方向和坦克连线之间的夹角
29            float cosa=(float) Math.acos((x1*x2+y1*y2+z1*z2)/(distance1*1));
30            if(cosa<Lock_angle){                             //如果其夹角在一定范围
31                if(Lock_tanke!=null){                        //如果之前有别的坦克被锁定
32                    Lock_tanke.this_isno_Lock=false;         //则标志其不被锁定
33                }
34                this.this_isno_Lock=true;                    //标志自己被锁定
35                minimumdistance=distance1;                   //最小距离设置为该距离
36                nx=x1;ny=y1+20;nz=z1;                        //记录发射子弹方向向量
37                isno_Lock=true;                              //已经被锁定
38                Lock_tanke=this;                             //自己被锁定
39            }else{
40                this_isno_Lock=false;                        //否则自己标识为未被锁定
41            }}
```

- 第2-14行为根据飞机位置计算代表该坦克生命值的矩形框的朝向。

- 第15-41行为计算其是否被锁定，其原理为每次查看其是否在一定距离范围内，如果是则查看其与飞机的连线方向和飞机飞行方向的夹角是否在一定范围内，在则说明被锁定。每次锁定后记录当前锁定的距离，并记录此时飞机与坦克连线的方向向量。

（4）最后开发的是该类中的坦克发射炮弹方法tank_go，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/tankemodel目录下的TanKe.java。

```
public void tank_go() {
        calculateBillboardDirection();          //计算目标生命值（血）显示矩形的标志版朝向
        float curr_planeX=PLANE_X;              //获取飞机的位置
        float curr_planeY=PLANE_Y;
```

```
        float curr_planeZ=PLANE_Z;
        float curr_distance=(curr_planeX-barrel_bottom_position[0])*(curr_planeX-
            barrel_bottom_position[0])+(curr_planeY-barrel_bottom_position[1])*
            (curr_planeY-barrel_bottom_position[1])+(curr_planeZ-barrel_bottom_
            position[2])*
            (curr_planeZ-barrel_bottom_position[2]);//计算飞机和坦克之间的距离
            if(curr_distance>TANK_MAX_DISTANCE*TANK_MAX_DISTANCE)    {
                return;                             //如果超出坦克的范围,那么直接返回
            }
            float curr_y_span=curr_planeY-barrel_bottom_position[1]; //计算高度差
            if(curr_y_span<=0){                     //如果小于 0,那么直接返回
                return;
            }
            curr_distance=(float) Math.sqrt(curr_distance); //计算距离
            float curr_elevation=(float) Math.toDegrees( Math.asin(curr_y_span/curr_
            distance));
            tank_barrel_elevation=curr_elevation;   //计算得到仰角
            float curr_x_span=curr_planeX-barrel_bottom_position[0]; //根据反正切计算方向角
            float curr_z_span=curr_planeZ-barrel_bottom_position[2];
            float curr_direction=(float)Math.toDegrees(Math.atan(curr_x_span/curr
            _z_span));
            if(curr_x_span==0&&curr_z_span==0){     //得到方向角
                tank_barrel_direction=curr_direction=0;
            }else if(curr_z_span>=0){
                tank_barrel_direction=curr_direction=curr_direction+180;
            }else    {
                tank_barrel_direction=curr_direction;
            }
            if(System.nanoTime()-oldTime>2000000000) { //如果可以发射炮弹, 每隔 2s 发射一次
                bomb_position_init[0]=(float) (barrel_bottom_position[0]-
                                                    //计算炮弹发射的初始位置
                    Math.cos(Math.toRadians(tank_barrel_elevation))*
                    Math.sin(Math.toRadians(tank_barrel_direction))*barrel_length);//X
                bomb_position_init[1]=(float) (barrel_bottom_position[1]+Math.sin(
                    Math.toRadians(tank_barrel_elevation))*barrel_length);//Y
                bomb_position_init[2]=(float)  (barrel_bottom_position[2]-Math.cos(
                                                    //发射炮弹
                    Math.toRadians(tank_barrel_elevation))*
                Math.cos(Math.toRadians(tank_barrel_direction))*barrel_length);//Z
                tank_bomb_List.add(new BombForControl(gv,bomb_ball,
                                                    //向炮弹列表中添加此炮弹
                    bomb_position_init,tank_barrel_elevation,tank_barrel_direction));
                gv.activity.playSound(1,0);         //播放发射炮弹声音
                oldTime=System.nanoTime();          //记录此次发射炮弹的时间
            }}
```

说明

该方法为实时改变炮管的朝向，使其始终朝向飞机，当飞机飞行到其一定的距离范围内时，每隔两秒发射一枚炮弹。对于高射炮类、敌机类和军火库类，其原理和坦克类差不多，其中军火库类只少了发射炮弹的功能，敌机类中，当敌机被锁定时，敌机朝着飞机飞行，本章就不再对这3个类进行详细介绍了，请读者自行查阅随书光盘中的源代码。

22.5.5　山地绘制类LandForm

该类为地形的绘制类，存在两种地形：用来搭建岛边沿的地形和使用灰度图加载而来的地形。第一种使用程序生成其高度数组然后卷绕三角形而得，第二种使用灰度图加载出地形的高度数组然后卷绕三角形而得，该类的详细代码如下。



代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/commonObject目录下的LandForm.java。

```
package com.bn.commonObject;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘中的源代码
public class LandForm{
    int mProgram;                                              //自定义渲染管线程序 id
    int muMVPMatrixHandle;                                     //总变换矩阵引用
    int maPositionHandle;                                      //顶点位置属性引用
    int maTexCoorHandle;                                       //顶点纹理坐标属性引用
    int uTuCengTexHandle;                                      //土层纹理属性引用
    int uCaoDiTexHandle;                                       //草地纹理属性引用
        int uShiTouTexHandle;                                  //石头纹理属性引用
        int uShanDingTexHandle;                                //山顶纹理属性引用
        int muHightHandle;                                     //渐变高度
        int muHightspanHandle;
        int uLandFlagHandle;                    //不同类型的山的标志，如果为 1 表示地面上的山
        private FloatBuffer   mVertexBuffer;                   //顶点坐标数据缓冲
        private FloatBuffer   mTextureBuffer;                  //顶点着色数据缓冲
        int vCount;                                            //顶点数量
        private boolean ishuidutu=false;                       //是灰度图还是程序生成
        public LandForm(int terrainId,int mProgram){
         this.mProgram=mProgram;
         if(terrainId==3||terrainId==5||terrainId==6){         //如果是陆地上的山,则不绘制倒影
             ishuidutu=true;
         }
         initVertexData(terrainId);                            //初始化数据
         initShader();                                         //初始化 shader
    }
    public void initVertexData(int terrainId) {                //初始化数据方法
    ……//此次为初始化数据方法，此处就不对其进行介绍了，请读者自行查阅随书光盘中的源代码
     }
     public void initShader(){                                 //初始化 shader
      maPositionHandle = GLES20.glGetAttribLocation(mProgram, "aPosition");
                                                               //顶点位置属性引用 id
      maTexCoorHandle= GLES20.glGetAttribLocation(mProgram, "aTexCoor");
                                                               //纹理坐标属性引用 id
      muMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
                                                               //总变换矩阵引用 id
      uShanDingTexHandle=GLES20.glGetUniformLocation(mProgram, "sTextureShanDing");
                                                               //山地的纹理图
      uTuCengTexHandle=GLES20.glGetUniformLocation(mProgram, "sTextureTuCeng");
                                                               //山地过程纹理
```

```
40    uCaoDiTexHandle=GLES20.glGetUniformLocation(mProgram, "sTextureCaoDi");
                                                  //总的使用 3 个纹理
41    muHightHandle=GLES20.glGetUniformLocation(mProgram, "height");  //第一个渐变高度
42    muHightspanHandle=GLES20.glGetUniformLocation(mProgram, "height_span");
                                                  //渐变高度
43    uShiTouTexHandle=GLES20.glGetUniformLocation(mProgram, "sTextureShiTou"); //纹理
44    uLandFlagHandle=GLES20.glGetUniformLocation(mProgram, "land_flag");  //
45    }
46    public void drawSelf(int landFlag,int tex_terrain_shandingId,
47       int texTuCengId,int texCaoDiId,int texShiTouId,float height,float height_span){
49          GLES20.glUseProgram(mProgram);            //制定使用某套 shader 程序
50          GLES20.glUniformMatrix4fv(muMVPMatrixHandle,  //将最终变换矩阵传入 shader 程序
51                1, false, MatrixState.getFinalMatrix(), 0);
52          GLES20.glUniform1f(muHightHandle, height);        //传入渐变的高度
53          GLES20.glUniform1f(muHightspanHandle, height_span);    //传入渐变的高度
54          GLES20.glUniform1i(uLandFlagHandle, landFlag);    //传入山的标志
55          GLES20.glVertexAttribPointer(                 //为画笔指定顶点位置数据
56                maPositionHandle, 3, GLES20.GL_FLOAT,
57                false,3*4,  mVertexBuffer);
58          GLES20.glVertexAttribPointer(              //为画笔指定顶点纹理坐标数据
59                maTexCoorHandle, 2, GLES20.GL_FLOAT,
60                false, 2*4, mTextureBuffer);
61          GLES20.glEnableVertexAttribArray(maPositionHandle);    //允许顶点位置数据数组
62          GLES20.glEnableVertexAttribArray(maTexCoorHandle);    //允许纹理坐标数据
63          GLES20.glActiveTexture(GLES20.GL_TEXTURE0);          //绑定纹理，4 幅纹理
64          GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, texTuCengId);
65          GLES20.glActiveTexture(GLES20.GL_TEXTURE1);
66          GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, texCaoDiId);
67          GLES20.glActiveTexture(GLES20.GL_TEXTURE2);
68          GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, texShiTouId);
69          GLES20.glActiveTexture(GLES20.GL_TEXTURE3);
70          GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, tex_terrain_shandingId);
71          GLES20.glUniform1i(uTuCengTexHandle, 0);
72          GLES20.glUniform1i(uCaoDiTexHandle, 1);
73          GLES20.glUniform1i(uShiTouTexHandle, 2);
74          GLES20.glUniform1i(uShanDingTexHandle, 3);
75          if(!ishuidutu){                                  //如果是用灰度图生成的
76             GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vCount); //执行绘制
77          }else{
78             GLES20.glDrawArrays(GLES20.GL_TRIANGLE_STRIP, 0, vCount);//执行绘制
79          }}}
```

- 第19-26行该类的构造器，传入高度数组的下标，调用初始化数据方法初始化数据。
- 第27-29行为初始化数据方法，其原理是根据高度数组卷绕三角形，在卷绕需要倒影的山地时，使其在y轴的负方向对称的卷绕对应的三角形，即可实现倒影的效果。而对应该方法中的详细代码，本章也不对其进行介绍，请读者自行查阅随书光盘中的源代码。
- 第30-45行为初始化着色器方法，根据着色器中对应成员变量，得到其位置属性id，然后再绘制时，根据id传入对应数据。
- 第46-79行为主绘制方法，对对应的属性id传入对应的数据即可，最后执行绘制。

22.5.6 颜色圆面类CircleForDraw

该类为根据传入的颜色值绘制一个圆面,主要用于绘制飞机选择场景中的展台圆面和用于构建高射炮中的对应圆面，其详细代码如下。

代码位置：见随书光盘中源代码/第22

章/AircraftAttack/src/com/bn/gameView目录下的CircleForDraw.java。

```java
package com.bn.gameView;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘中的源代码
public class CircleForDraw{
    float taizihight=30;
    int mProgram;                                        //自定义渲染管线程序 id
    int muMVPMatrixHandle;                               //总变换矩阵引用 id
    int maPositionHandle;                                //顶点位置属性引用 id
    int maColorR;                                        //颜色值的 R 分量引用 id
    int maColorG;                                        //颜色值的 G 分量引用 id
        int maColorB;                                    //颜色值的 B 分量引用 id
        int maColorA;                                    //颜色值的 A 分量引用 id
        private static FloatBuffer   mVertexBuffer;      //顶点坐标数据缓冲
        private static int vCount;                       //顶点数量
        float r;                                         //颜色值的 R 分量
        float g;                                         //颜色值的 G 分量
        float b;                                         //颜色值的 B 分量
        Light_Tower taizi;                               //展台圆柱
        public CircleForDraw (int mProgram,
             float angleSpan,                            //切分角度
             float radius,                               //圆半径
             float[]color,                               //颜色数组
             int mProgramlitht) {
         this.r=color[0];                                //得到颜色值
         this.g=color[1];
         this.b=color[2];
         initVertexData(angleSpan,radius);               //初始化数据
         initShader(mProgram);                           //初始化 shader 程序
         taizi=new Light_Tower(radius,radius,taizihight,1);    //展台圆柱
         taizi.initShader(mProgramlitht);                      //初始化其着色器
        }
        public static void initVertexData(float angleSpan, float radius){
                                                               //初始化数据方法
         vCount=1+(int)(360/angleSpan)+1;                      //顶点的个数
         float[] vertices=new float[vCount*3];                 //初始化顶点数组
         vertices[0]=0;
         vertices[1]=0;
         vertices[2]=0;
         int vcount=3;                                         //当前顶点坐标索引
         for(float angle=0;angle<=360;angle=angle+angleSpan){ //循环得到点坐标
             double angleRadian=Math.toRadians(angle);
             vertices[vcount++]=radius*(float)Math.cos(angleRadian);
             vertices[vcount++]=radius*(float)Math.sin(angleRadian);
             vertices[vcount++]=0;
```

```
43        }
44           ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
45           vbb.order(ByteOrder.nativeOrder());              //设置字节顺序
46           mVertexBuffer = vbb.asFloatBuffer();             //转换为 float 型缓冲
47           mVertexBuffer.put(vertices);                     //向缓冲区中放入顶点坐标数据
48           mVertexBuffer.position(0);                       //设置缓冲区起始位置
49        }
50        public void initShader(int mProgramIn){             //初始化 shader 程序方法
51           mProgram = mProgramIn;
52           maPositionHandle = GLES20.glGetAttribLocation(mProgram, "aPosition");
                                                              //获得对应变量的属性 id
53           muMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
54           maColorR=GLES20.glGetUniformLocation(mProgram, "colorR");
55           maColorG=GLES20.glGetUniformLocation(mProgram, "colorG");
56           maColorB=GLES20.glGetUniformLocation(mProgram, "colorB");
57           maColorA=GLES20.glGetUniformLocation(mProgram, "colorA");
58        }
59        public void drawSelf(float alpha,int texId){
60           GLES20.glUseProgram(mProgram);                   //指定使用那套 shader 程序
61           GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, false, MatrixState.
             getFinalMatrix(), 0);
62           GLES20.glVertexAttribPointer (                   //为画笔指定顶点位置数据
63                 maPositionHandle, 3, GLES20.GL_FLOAT, false, 3*4, mVertexBuffer );
64           GLES20.glEnableVertexAttribArray(maPositionHandle);
                                                              //允许顶点位置、法向量数据数组
65           GLES20.glUniform1f(maColorR , r);                //传入 r、g、b、a 值
66           GLES20.glUniform1f(maColorG , g);
67           GLES20.glUniform1f(maColorB , b);
68           GLES20.glUniform1f(maColorA , alpha);
69           GLES20.glDrawArrays(                             //执行绘制
70                 GL10.GL_TRIANGLE_FAN,                      //以 TRIANGLE_FAN 方式填充
71                 0, vCount);
72        }}
```

- 第17-30行为该类的构造器，创建该对象时，需传入需要使用的着色器程序的id、圆的切分角度、圆的半径和该圆面的颜色数组。
- 第31-49行为初始化顶点数据方法，该方法根据切分的角度和半径来卷绕三角形，最后根据顶点数组创建顶点缓冲数据。
- 第59-58行为根据当前使用的着色器程序来获得对应变量的id，在绘制时根据其属性id传入对应的值，最后即可执行绘制。
- 第59-72行为具体的绘制方法，需对其传入总变换矩阵、顶点数据和r、g、b、a颜色值，最后执行绘制即可。

22.5.7　星空类SkyNight

该类为通过在指定位置绘制一些白色圆点来构造一个黑夜里的星空，而整个黑夜星空的实现原理为，首先在外层绘制一个使用黑色纹理的天空，然后里面使用该类绘制一个星空，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/src/com/bn/commonObject目录下的SkyNight.java。

```
package com.bn.commonObject;
……//此处省略了部分类和包的引入代码，读者可自行查阅光盘中的源代码
public class SkyNight {
    float skyR=30.0f;                                        //天球半径
    private FloatBuffer   mVertexBuffer;                     //顶点坐标数据缓冲
   int vCount=0;                                             //星星数量
   float scale;                                              //星星尺寸
   int mProgram;                                             //自定义渲染管线程序 id
   int muMVPMatrixHandle;                                    //总变换矩阵引用 id
       int maPositionHandle;                                 //顶点位置属性引用 id
       int uPointSizeHandle;                                     //顶点尺寸参数引用
       public SkyNight(float scale,int vCount,float skyR){       //构造器
        this.skyR=skyR;                                          //星空半径
        this.scale=scale;                                        //星星尺寸
        this.vCount=vCount;                                      //星星数量
        initVertexData();                                        //初始化星星位置数据
       }
       public void initVertexData(){                             //初始化顶点坐标的方法
           float vertices[]=new float[vCount*6];
           for(int i=0;i<vCount;i++){                        //随机产生每个星星的 x、y、z 坐标
             double angleTempJD=Math.PI*2*Math.random();         //随机产生一个经度值
             double angleTempWD=Math.PI/3*(Math.random());       //随机产生一个纬度值
             vertices[i*6]=(float)(skyR*Math.cos(angleTempWD)*Math.sin(angleTempJD));
             vertices[i*6+1]=(float)(skyR*Math.sin(angleTempWD));
             vertices[i*6+2]=(float)(skyR*Math.cos(angleTempWD)*Math.cos(angleTempJD));
             vertices[i*6+3]=(float)(skyR*Math.cos(angleTempWD)*Math.sin(angleTempJD));
             vertices[i*6+4]=-(float)(skyR*Math.sin(angleTempWD));
                                                             //水面下的倒影，一一对应
             vertices[i*6+5]=(float)(skyR*Math.cos(angleTempWD)*Math.cos(angleTempJD));
           }
           ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length*4);
                                                             //创建顶点坐标数据缓冲
           vbb.order(ByteOrder.nativeOrder());               //设置字节顺序
           mVertexBuffer = vbb.asFloatBuffer();              //转换为 int 型缓冲
           mVertexBuffer.put(vertices);                      //向缓冲区中放入顶点坐标数据
           mVertexBuffer.position(0);                        //设置缓冲区起始位置
       }
       public void initShader(int mProgram){                 //初始化 shader
           this.mProgram = mProgram;
           maPositionHandle = GLES20.glGetAttribLocation(mProgram, "aPosition");
           muMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
           uPointSizeHandle = GLES20.glGetUniformLocation(mProgram, "uPointSize");
       }
       public void drawSelf(){
            GLES20.glUseProgram(mProgram);                       //指定使用某套 shader 程序
           GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, false, MatrixState.
           getFinalMatrix(), 0);
```

```
45          GLES20.glUniform1f(uPointSizeHandle, scale);  //将顶点尺寸传入Shader程序
47          GLES20.glVertexAttribPointer(               //为画笔指定顶点位置数据
48                maPositionHandle, 3, GLES20.GL_FLOAT, false, 3*4, mVertexBuffer);
49          GLES20.glEnableVertexAttribArray(maPositionHandle);    //允许顶点位置数据数组
50          GLES20.glDrawArrays(GLES20.GL_POINTS, 0, vCount*2); //绘制星星点
51      }}
```

- 第18-35行为初始化星星的位置数组，其原理是随机产生一个角度，然后根据半径和产生的随机角度计算得到一个位置坐标，而将其*y* 坐标置反即可得到其倒影星星的位置，最后创建其顶点坐标数据缓冲。
- 第36-51行分别为初始化着色器方法和具体绘制方法，详细情况请见注释。

22.5.8　爆炸效果绘制类DrawBomb

该类为绘制一个爆炸效果，其原理是将一个纹理矩形不断放大绘制，最后消失。每次需要绘制一个爆炸效果时，将该类的一个对象添加进爆炸效果列表，其放大绘制的整个过程由其在自身的绘制方法中处理，不需要在别的线程中对其进行额外的操作，当放大到一定程度时，从其列表中删除该对象即可，其详细代码如下。



代码位置：见随书光盘中源代码/第22

章/AircraftAttack/src/com/bn/commonObject目录下的DrawBomb.java。

```
1package com.bn.commonObject;;
2……//此处省略了该类中相关的常量和静态变量的声明，请读者自行查阅随书所带光盘中的源代码
3public class DrawBomb{
4     float x,y,z;                                              //位置坐标
5     float xAngle,yAngle,zAngle;                               //绕对应轴的旋转角度
6     int index;                                                //当前的放大系数
7     TextureRect recw;                                         //矩形引用
8     public DrawBomb(TextureRect recw,float x,float y,float z) {   //构造器
9          this.x=x;
10              this.y=y;
11              this.z=z;
12              this.recw=recw;
13         }
14         public void drawSelf(){                          //绘制方法
15              index++;                                    //放大系数每刷一帧自加1
16              if(index>15){                               //如果到了第15帧
17                   try{
18                        Iterator<DrawBomb> ite=baoZhaList.iterator();//得到该列表的迭代器
19                        while(ite.hasNext()){
20                             if(ite.next()==this) {
21                                  ite.remove();           //从列表中删除此爆炸效果
22                                  return;
23                             }}}
24                   catch(Exception e){
25                        e.printStackTrace();
26                   }
27                   return;
28              }
29              calculateBillboardDirection();             //计算其朝向
30              MatrixState.pushMatrix();                   //保护现场
31              MatrixState.translate(x, y, z);             //推动坐标到该爆炸效果的位置
32              MatrixState.rotate(xAngle, 1, 0, 0);        //绕X轴旋转对应的角度
33              MatrixState.rotate(yAngle, 0, 1, 0);        //绕Y轴旋转对应的角度
34              MatrixState.rotate(zAngle, 0, 0, 1);        //绕Z轴旋转对应的角度
35              MatrixState.scale(index/3%5*0.8f, index/3%5*0.8f, index/3%5*0.8f);
                                                            //设置放大系数
36              recw.drawSelf(baoZhaXiaoguo2);              //绘制该爆炸效果
37              MatrixState.popMatrix();                    //恢复现场
38         }
39         public void calculateBillboardDirection(){  //根据摄像机位置计算爆炸纹理面朝向
40              float xspan=x-cx;                   //得到摄像机位置，到爆炸效果之间的方向向量
41              float zspan=z-cz;
42              float yspan=y-cy;
43              if(zspan<=0) {                              //如果z方向小于0
44                   yAngle=(float)Math.toDegrees(Math.atan(xspan/zspan));
                                                            //根据反正切值得到对应角度
45              }else{
46                   yAngle=180+(float)Math.toDegrees(Math.atan(xspan/zspan));
                                                            //否则需要加180°
47              }
48              xAngle=(float)Math.toDegrees(Math.atan(yspan/Math.sqrt(xspan*xspan+
                zspan*zspan)));
49         }}
```

- 第14-38行为绘制方法，在该方法中，每刷一帧其对应变量自加1，当其绘制了15次后，需要将该爆炸效果从爆炸列表中删除，然后

根据绘制的次数换算出放大的系数来执行绘制。

- 第39-49行为计算该爆炸纹理矩形的朝向，每次绘制时需要其正对摄像机的方向。

22.5.9　数字绘制类NumberForDraw

该类为绘制需要的数字字符串，传入当前需要绘制的数字，然后绘制出来，其原理是构造10个纹理坐标不一样的矩形，每个矩形对应着相应的数字纹理坐标，其详细代码如下。

代码位置：见随书光盘中源代码/第22

章/AircraftAttack/src/com/bn/gameView目录下的Constantjava。

```
package com.bn.commonObject;
import com.bn.core.MatrixState;
public class NumberForDraw {
    public TextureRect[] number;                          //数字纹理矩形数组
    String scoreStr;                                      //数字字符串
    float width;                                          //每个数字的宽高
    float height;
    public NumberForDraw(int numberSize,float width,float height,int mProgram) {//构造器
        number=new TextureRect[numberSize];               //创建数字数据对象
            this.width=width;                             //得到宽高
            this.height=height;
            for(int i=0;i<numberSize;i++){                //生成10个数字的纹理矩形
```

```
13                 number[i]=new TextureRect(
14                     width,height,
15                 new float[]{                              //构造纹理坐标
16                     1f/numberSize*i,0,1f/numberSize*i,1, 1f/numberSize*(i+1),0,
17                     1f/numberSize*(i+1),1,
18                  },
19                  mProgram                                 //传入对应的 shader 程序 id
20               ); }}
21       public void drawSelf(String score,int texId) {   //传入数字和纹理 id，右对齐
22           scoreStr=score;                               //得到数字字符串
23           MatrixState.pushMatrix();                     //保护现场
24           MatrixState.translate(-scoreStr.length()*width, 0, 0);
                                                           //根据数字个数沿 X 轴推动一定距离
25           for(int i=0;i<scoreStr.length();i++){         //将得分中的每个数字字符进行绘制
26               char c=scoreStr.charAt(i);                //得到对应的数字
27             MatrixState.translate(width, 0, 0);         //沿 X 轴推动数字的宽度
28             number[c-'0'].drawSelf(texId);              //执行绘制
29           }
30           MatrixState.popMatrix();                      //恢复现场
31       }
32       public void drawSelfLeft(String score,int texId) {   //传入数字和纹理坐标，左对齐
33           scoreStr=score;
34           MatrixState.pushMatrix();                     //保护现场
35           for(int i=0;i<scoreStr.length();i++){         //将得分中的每个数字字符绘制
36               char c=scoreStr.charAt(i);                //得到对应的数字
37             MatrixState.translate(width, 0, 0);         //沿 X 轴推动一个数字宽度的距离
38             number[c-'0'].drawSelf(texId);              //执行绘制此数字
39           }
40           MatrixState.popMatrix();                      //恢复现场
41       }}
```

- 第8-20行为该类的构造器，在该构造器中，创建10个对应数组纹理坐标的纹理矩阵，因为该数字纹理图是0～9的数字。
- 第21-41行分别为绘制右对齐和左对齐到当前点的数字，传入一个数字字符串，然后逐个取出对应字符进行绘制即可。

22.6　该游戏中的着色器

上一节对游戏界面绘制的相关类进行了介绍，下面就对该游戏中用到的相关着色器进行介绍，本游戏中用到的着色器有加载界面着色器、地形着色器、按钮着色器、星空着色器、生命值矩形着色器和颜色着色器。下面就对其详细介绍。

22.6.1　加载界面着色器

着色器分为顶点着色器和片元着色器，下面就对其代码开发一一进行介绍。

（1）首先开发的是加载界面的顶点着色器，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/assets目录下的vertex_tex_only.sh。

```
uniform mat4 uMVPMatrix;                              //总变换矩阵
attribute vec3 aPosition;                             //顶点位置
attribute vec2 aTexCoor;                              //顶点纹理坐标
varying vec2 vTextureCoord;                           //用于传递给片元着色器的变量
void main(){
  gl_Position = uMVPMatrix * vec4(aPosition,1);       //根据总变换矩阵计算此顶点位置
  vTextureCoord = aTexCoor;                           //将接收的纹理坐标传递给片元着色器
}
```

说明

该顶点着色器的作用主要为根据顶点位置和总变换矩阵计算gl_Position，每顶点执行一次。

（2）完成该顶点着色器的开发后，下面开发的是加载界面的片元着色器，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/assets目录下的frag_tex_only.sh。

```
precision mediump float;
varying vec2 vTextureCoord;                                //接收从顶点着色器过来的参数
uniform sampler2D sTexture;                                //纹理内容数据
void main() {
  gl_FragColor = texture2D(sTexture, vTextureCoord);       //给此片元从纹理中采样出颜色值
}
```

说明

该片元着色器的作用主要为根据从顶点着色器传递过来的参数vTextureCoord和从Java代码部分传递过来的sTexture计算片元的最终颜色值，每片元执行一次。

22.6.2 地形着色器

地形着色器是使用过程纹理技术，根据其坐标位置使用对应的纹理，在相接的地方使用渐变，使其更有真实感。

（1）首先开发地形着色器的顶点着色器，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/assets目录下的vertex_landform.sh。

```
uniform mat4 uMVPMatrix;                              //总变换矩阵
attribute vec3 aPosition;                             //顶点位置
attribute vec2 aTexCoor;                              //顶点纹理坐标
varying vec2 vTextureCoord;                           //用于传递给片元着色器的变量
varying float vertexHeight;                           //接受顶点的高度值
void main(){
   gl_Position = uMVPMatrix * vec4(aPosition,1);      //根据总变换矩阵计算此顶点位置
   vTextureCoord = aTexCoor;                          //将接收的纹理坐标传递给片元着色器
   vertexHeight = aPosition.y;                        //将该顶点的高度传入片元着色器
   }
```

说明

该顶点着色器的作用主要为根据顶点位置和总变换矩阵计算gl_Position，并将从Java代码传入的纹理坐标数据与该顶点的高度值传入到相应片元着i器中。

（2）完成地形着色器顶点着色器的开发后，下面开发地形着色器的片元着色器，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/assets目录下的frag_tex_landform.sh。

```
precision mediump float;
varying vec2 vTextureCoord;                                  //接收纹理坐标参数
varying float vertexHeight;                                  //接收顶点的高度值
uniform sampler2D usTextureTuCeng;                           //纹理内容数据——土层
uniform sampler2D usTextureCaoDi;                            //纹理内容数据——草地
uniform sampler2D usTextureShiTou;                           //纹理内容数据——石头
uniform sampler2D usTextureShanDing;                         //纹理内容数据——山顶
uniform float uheight;                                       //第一个高度
uniform float uheight_span;                                  //渐变高度
   uniform int uland_flag;                                   //山的标志 1 为地面上的高山
   void main(){
       float height1=30.0;                                   //第一个渐变高度
       float height2=30.0;                                   //第二个渐变高度
       float height3=10.0;                                   //第三个渐变高度
       float land1=250.0;                                    //陆地山的过度高度
       float land2=400.0;                                    //陆地山的石头绘制高度
       vec4 finalColor0=vec4(0.3,0.3,0.3,0.5);               //黑色条的颜色
     vec4 finalColor1=texture2D(usTextureTuCeng, vTextureCoord);   //土层
     vec4 finalColor2=texture2D(usTextureCaoDi, vTextureCoord);    //草地
     vec4 finalColor3=texture2D(usTextureShiTou, vTextureCoord);   //石头
     vec4 finalColor4=texture2D(usTextureShanDing, vTextureCoord);//山顶纹理
     if(uland_flag==0){                                            //如果是搭建岛的山地
         if(abs(vertexHeight)<uheight){                            //第一个阶段
           float ratio=abs(vertexHeight)/uheight;                  //渐变比例
           finalColor3 *=(1.0-ratio);                              //颜色渐变比例
            finalColor0 *=ratio;                                   //得到纹理中颜色值
           gl_FragColor =finalColor3+ finalColor0;                 //得到最终颜色值
         }
         else if(abs(vertexHeight)>=uheight&&abs(vertexHeight)<=uheight+height1) {
                                                                   //第一个渐变高度
               float ratio=(abs(vertexHeight)-uheight)/height1;//土层颜色比例
               finalColor0 *=(1.0-ratio);                          //颜色的比例
               finalColor1 *=ratio;                                //得到最终采样颜色值
               gl_FragColor =finalColor1 + finalColor0;            //得到最终颜色
         }
         else if(abs(vertexHeight)>uheight+height1&&abs(vertexHeight)<=uheight_
         span-height2){
               gl_FragColor =finalColor1;                      //如果是第二个区域直接赋值
         }
         else if(abs(vertexHeight)>=uheight_span-height2&&abs(vertexHeight)<=
         uheight_span){
               float ratio=(abs(vertexHeight)-uheight_span+height2)/height2;
                                                               //第二个渐变区域
               finalColor1 *=(1.0-ratio);                      //采样颜色比例
               finalColor0 *=ratio;
```

```
42             gl_FragColor =finalColor1 + finalColor0;            //最终颜色
43         }
44         else if(abs(vertexHeight)>=uheight_span&&abs(vertexHeight)<=uheight_
           span+height3){
45             float ratio=(abs(vertexHeight)-uheight_span)/height3;//第三个渐变区域
46             finalColor0 *=(1.0-ratio);                          //采样颜色值
47             finalColor2 *=ratio;
48             finalColor0.a=0.2;
49             gl_FragColor =finalColor2 + finalColor0;            //得到最终颜色
50         } else{
51           gl_FragColor = finalColor2;                //否则直接赋值
52         }}else{                                      //否则为陆地上的山，只有两个渐变区域
53           if(abs(vertexHeight)<land1){               //草地层
54             gl_FragColor = finalColor2;
55           }
56           else if(abs(vertexHeight)>=land1&&abs(vertexHeight)<=land2){
                                                        //草地与山顶之间渐变区域
57             float ratio=(abs(vertexHeight)-land1)/(land2-land1);
58             finalColor2 *=(1.0-ratio);
59             finalColor4 *=ratio;
60             gl_FragColor =finalColor2 + finalColor4;    //最终颜色值
61           }else{                                        //山顶区域
62             gl_FragColor = finalColor4;
63           }}}
```

- 第22-52行为搭建岛的山地绘制时，其有6个区域，最下面土层区域、黑线渐变、中间石头纹理、石头渐变到黑线、黑线渐变到草地和草地。
- 第52-63行为陆地上的山地绘制时，只有3个区域，为草地区域、草地过度到山顶区域和山顶区域。

22.6.3 按钮着色器

按钮着色器是当玩家触摸到某个按钮时，将按钮的颜色变暗淡些，按钮的顶点着色器和加载界面着色器的顶点着色器是同一个，下面只介绍其片元着色器的开发，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/assets目录下的frag_button.sh。

```
precision mediump float;
varying vec2 vTextureCoord;                                    //接收从顶点着色器过来的参数
uniform sampler2D sTexture;                                    //纹理内容数据
uniform int uisButtonDown;                                     //按钮是否按下标志位
void main(){
  vec4 finalColor = texture2D(sTexture, vTextureCoord);        //给此片元从纹理中采样出颜色值
  if(uisButtonDown==1) {                                       //当前按钮按下
     gl_FragColor=finalColor*0.5;                              //使其颜色值变暗
  }else {                                                      //否则使用原本颜色
        gl_FragColor=finalColor;
     }}
```

说明

该片元着色器的作用为根据传入的uisButtonDown数据情况，改变该片元的颜色值。

22.6.4　星空着色器

星空着色器用来绘制天空中白色圆点，此处仅给出星空着色器顶点着色器的开发，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/assets目录下的vertex_xk.sh。

```
uniform mat4 uMVPMatrix;                                  //总变换矩阵
uniform float uPointSize;                                 //点尺寸
attribute vec3 aPosition;                                 //顶点位置
void main(){
  gl_Position = uMVPMatrix * vec4(aPosition,1);           //根据总变换矩阵计算此顶点位置
  gl_PointSize=uPointSize;                                //点的最终尺寸
}
```

说明

该顶点着色器中，从Java代码部分传入的uPointSize数据，最终赋值给gl_PointSize。gl_PointSize表示的是星空中每个点的大小。

22.6.5 生命值矩形着色器

该着色器根据当前物体的生命值来对应地变换颜色，使其生命值等效到一个红条上，在顶点着色器中需要将点的x 坐标和y 坐标传入片元着色器，所以此处只为大家介绍片元着色器，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/assets目录下的frag_xue.sh。

```
precision mediump float;
varying vec2 vTextureCoord;                                  //接收纹理坐标参数
varying float vertexHeight;                                  //接收顶点的高度值
varying float vertexwhidth;                                  //接收生命值的最左边位置
uniform sampler2D sTexture;                                  //纹理内容数据
uniform float ublood;                                        //接收生命值数据
void main(){
  vec4 finalColor=texture2D(sTexture, vTextureCoord);        //采样出颜色
      if(vertexwhidth<ublood&&vertexwhidth>-97.0&&vertexwhidth<97.0&&
         vertexHeight>-6.5&&vertexHeight<6.5){          //如果在此范围
         gl_FragColor = vec4(0.5,0.17,0.04,1.0);        //坐标小于生命值的地方绘制指定
      } else {
         gl_FragColor=finalColor;                       //否则绘制背景颜色
      }}
```

说明

该片元着色器的作用主要为根据从顶点着色器传入的vertexHeight和vertexwhidth数据设定一个范围，在该范围内声明值为一个颜色，在该范围外则为背景颜色。

22.6.6 颜色圆面着色器

该着色器需要在绘制时为片元着色器传入对应的颜色值，而该顶点着色器代码和其他顶点着色 器代码几乎没有区别，此处只介绍该着色器的片元着色器的开发，其详细代码如下。

代码位置：见随书光盘中源代码/第22章/AircraftAttack/assets目录下的frag_color.sh。

```
precision mediump float;
uniform float colorR;                                    //颜色值的 R 分量
uniform float colorG;                                    //颜色值的 G 分量
uniform float colorB;                                    //颜色值的 B 分量
uniform float colorA;                                    //颜色值的 A 分量
void main(){
        vec4 finalColor;
        finalColor=vec4(colorR,colorG,colorB,colorA);//根据传入的颜色值构造一个片元颜色
       gl_FragColor = finalColor;                        //给此片元颜色值
   }
```

说明

该片元着色器的作用为根据Java代码部分传入的颜色值信息确定该片元的最终颜色值。

22.7　游戏的优化与改进

本游戏采用OpenGL ES 2.0作为渲染引擎，虽然此游戏已经开发完毕，但是依然还有很多值得改进和提升的地方，笔者在此列出了一下几个方面。

- 游戏界面的优化。

没有哪一款游戏的界面不可以更加完美和绚丽的，所以对本游戏的界面，读者可自行根据自己的思想进行改进，使其更加完美。如地形的搭建，可以搭建出更具真实感的山地地图，飞机模型、飞机的贴图等都可以进一步地完善。

- 游戏玩法的完善。

此游戏的玩法还是较为单一，只有两种模式，而且对应模式下的玩法设计也不是很好，读者可自行完善，增加更多的玩法，使其更具吸引力。

- 游戏中物理引擎的完善。

本游戏中碰撞检测做的还是比较粗糙的，对子弹和炮弹的模拟也还不够真实，希望读者能对其升级，更加细致地去模拟真实世界的效果。